MIPS IV Instruction Set

Revision 3.2

September, 1995

Charles Price

Information on MIPS products is available electronically:

(a) Through the World Wide Web. Point your WWW client to:

http://www.mips.com

(b) Through ftp from the internet site "sgigate.sgi.com". Login as "ftp" or "anonymous" and then cd to the directory "pub/doc".

(c) Through an automated FAX service:

Inside the USA toll free: (800) 446-6477 (800-IGO-MIPS)

Outside the USA: (415) 688-4321 (call from a FAX machine)

MIPS Technologies, Inc.
2011 N. Shoreline Blvd.
Mountain View, CA 94039-7311
Phone: USA toll free: (800) 998-6477 Outside USA: (415) 933-6477

MIPS IV Instruction Set

CPU Instruction Set

FPU Instruction Set

List of Figures

List of Tables

Revision History

2.0 (Jan 94): First General Release

This version contained incorrect definitions for MSUB and NMSUB. It did not contain the RECIP and RSQRT instructions. It contained incomplete or erroneous information for LL, LLD, SC, SCD, SYNC, PREF, and PREFX.

All copies of this version of the document should be destroyed

2.2 (Jul 94): Mandatory Replacement of Rev 2.0

This version should probably have been 3.0 since it is a major content change.

This version is issued with no known errors. It includes the late changes to the MIPS IV definition including the reintroduction of RECIP and RSQRT and the definition of the multiply-accumulate instructions as unfused (rounded) operations.

3.0 (Oct 94):

Add itemized instruction lists in the discussion of instruction functional groups.

Add a more complete description of FPU operation

Correct problems discovered with Revision 2.2.

3.1 (Jan 95):

Correct minor problems discovered with Revision 3.0.

3.2 (Sep 95):

Revise the opcode encoding tables significantly.

Correct minor problems discovered with Revision 3.1.

Changes from previous revision:

Changes are generally marked by change bars in the outer margin of the page -- just like the bar to the side of this line. Minor corrections to punctuation and spelling are neither marked with change bars nor noted in this list. Some changes in figures are not marked by change bars due to limitations of the publishing tools.

CVT.D.fmt Instruction

Change the architecture level for the CVT.D.L version of the instruction
from:
to: MIPS III

CVT.S.fmt Instruction

Change the architecture level for the CVT.S.L version of the instruction

from:

to: MIPS III

LWL Instruction

In the example in Fig. A-4 the sign extension "After executing `LWL $24,2($0)`" should be changed

from: no cng or sign ext

to: sign bit (31) extend.

The information in the tables later in the instruction description is correct.

MOVF Instruction

Change the name of the constant value in the function field

from: MOVC

to: MOVCI

There is a corresponding change in the FPU opcode encoding table in section B.12 with opcode=*SPECIAL* and function=MOVC, changing the value to MOVCI.

MOVF.fmt Instruction

Change the name of the constant value in the function field

from: MOVC

to: MOVCF

There is a corresponding change in the FPU opcode encoding table in section B.12 with opcode=*COP1*, fmt = *S* or *D*, and function=MOVC, changing the value to MOVCI.

MOVF Instruction

Change the name of the constant value in the function field

from: MOVC

to: MOVCI

There is a corresponding change in the FPU opcode encoding table in section B.12 with opcode=*SPECIAL* and function=MOVC, changing the value to MOVCI.

MOVT.fmt Instruction

Change the name of the constant value in the function field

from: MOVC

to: MOVCF

There is a corresponding change in the FPU opcode encoding table in section B.12 with opcode=*COP1*, fmt = *S* or *D*, and function=MOVC, changing the value to MOVCI.

CPU Instruction Encoding tables

Revise the presentation of the opcode encoding in section A 8 for greater clarity when considering different architecture levels or operating a MIPS III or MIPS IV processor in the MIPS II or MIPS III instruction subset modes.

There is a separate encoding table for each architecture level. There is a table of the MIPS IV encodings showing the architecture level at which each opcode was first defined and subsequently modified or extended. There is a separate table for each architecture revision I→II, II→III, and III→IV showing the changes made in that revision.

FPU Instruction Encoding tables

Revise the presentation of the opcode encoding in section B.12 for greater clarity when considering different architecture levels or operating a MIPS III or MIPS IV processor in the MIPS II or MIPS III instruction subset modes.

There is a separate encoding table for each architecture level. There is a table of the MIPS IV encodings showing the architecture level at which each opcode was first defined and subsequently modified or extended. There is a separate table for each architecture revision I→II, II→III, and III→IV showing the changes made in that revision.

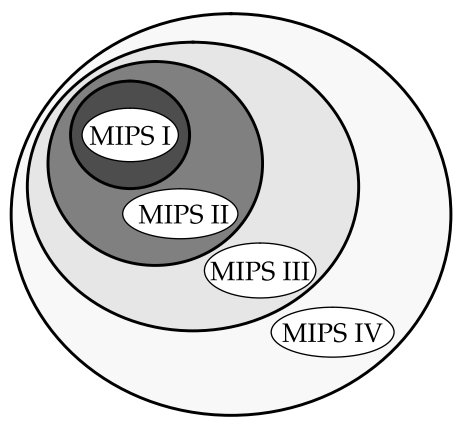
CPU Instruction Set

A

A 1 Introduction

This appendix describes the instruction set architecture (ISA) for the central processing unit (CPU) in the MIPS IV architecture. The CPU architecture defines the non-privileged instructions that execute in user mode. It does not define privileged instructions providing processor control executed by the implementation-specific System Control Processor. Instructions for the floating-point unit are described in Appendix B.

The original MIPS I CPU ISA has been extended in a backward-compatible fashion three times. The ISA extensions are inclusive as the diagram illustrates; each new architecture level (or version) includes the former levels. The description of an architectural feature includes the architecture level in which the feature is (first) defined or extended. The feature is also available in all later (higher) levels of the architecture.



MIPS Architecture Extensions

The practical result is that a processor implementing MIPS IV is also able to run MIPS I, MIPS II, or MIPS III binary programs without change.

The CPU instruction set is first summarized by functional group then each instruction is described separately in alphabetical order. The appendix describes the organization of the individual instruction descriptions and the notation used in them (including FPU instructions). It concludes with the CPU instruction formats and opcode encoding tables.

A 2 Functional Instruction Groups

CPU instructions are divided into the following functional groups:

- Load and Store
- ALU
- Jump and Branch
- Miscellaneous
- Co processor

A 2.1 Load and Store Instructions

Load and store instructions transfer data between the memory system and the general register sets in the CPU and the coprocessors. There are separate instructions for different purposes: transferring various sized fields, treating loaded data as signed or unsigned integers, accessing unaligned fields, selecting the addressing mode, and providing atomic memory update (read-modify-write).

Regardless of byte ordering (big- or little-endian), the address of a halfword, word, or doubleword is the smallest 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.

Except for the few specialized instructions listed in Table A-4, loads and stores must access naturally aligned objects. An attempt to load or store an object at an address that is not an even multiple of the size of the object will cause an Address Error exception.

Load and store operations have been added in each revision of the architecture:

MIPS II

- 64-bit coprocessor transfers
- atomic update

MIPS III

- 64-bit CPU transfers
- unsigned word load for CPU

MIPS IV

- register + register addressing mode for FPU

Tables A-1 and A-2 tabulate the supported load and store operations and indicate the MIPS architecture level at which each operation was first supported. The instructions themselves are listed in the following sections.

Table A-1 Load/Store Operations Using Register + Offset Addressing Mode.

	CPU			coprocessor (except 0)	
Data Size	**Load Signed**	**Load Unsigned**	**Store**	**Load**	**Store**
byte	I	I	I		
halfword	I	I	I		
word	I	III	I	I	I
doubleword	III		III	II	II
unaligned word	I		I		
unaligned doubleword	III		III		
linked word (atomic modify)	II		II		
linked doubleword (atomic modify)	III		III		

Table A-2 Load/Store Operations Using Register + Register Addressing Mode.

	floating-point coprocessor only	
Data Size	**Load**	**Store**
word	IV	IV
doubleword	IV	IV

A 2.1.1 Delayed Loads

The MIPS I architecture defines delayed loads; an instruction scheduling restriction requires that an instruction immediately following a load into register *Rn* cannot use *Rn* as a source register. The time between the load instruction and the time the data is available is the "load delay slot". If no useful instruction can be put into the load delay slot, then a null operation (assembler mnemonic NOP) must be inserted.

In MIPS II, this instruction scheduling restriction is removed. Programs will execute correctly when the loaded data is used by the instruction following the load, but this may require extra real cycles. Most processors cannot actually load data quickly enough for immediate use and the processor will be forced to wait until the data is available. Scheduling load delay slots is desirable for performance reasons even when it is not necessary for correctness.

A 2.1.2 CPU Loads and Stores

There are instructions to transfer different amounts of data: bytes, halfwords, words, and doublewords. 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 A-3 Normal CPU Load/Store Instructions

Mnemonic	Description	Defined in
LB	Load Byte	MIPS I
LBU	Load Byte Unsigned	I
SB	Store Byte	I
LH	Load Halfword	I
LHU	Load Halfword Unsigned	I
SH	Store Halfword	I
LW	Load Word	I
LWU	Load Word Unsigned	III
SW	Store Word	I
LD	Load Doubleword	III
SD	Store Doubleword	III

Unaligned words and doublewords can be loaded or stored in only two instructions by using a pair of special instructions. 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. MIPS I, though it prohibits other use of loaded data in the load delay slot, permits LWL and LWR instructions targeting the same destination register to be executed sequentially. Store instructions select the correct bytes from a source register and update only those bytes in an aligned memory word (or doubleword).

Table A-4 Unaligned CPU Load/Store Instructions

Mnemonic	Description	Defined in
LWL	Load Word Left	MIPS I
LWR	Load Word Right	I
SWL	Store Word Left	I
SWR	Store Word Right	I
LDL	Load Doubleword Left	III
LDR	Load Doubleword Right	III
SDL	Store Doubleword Left	III
SDR	Store Doubleword Right	III

A 2.1.3 Atomic Update Loads and Stores

There are paired instructions, Load Linked and Store Conditional, that can be used to perform atomic read-modify-write of word and 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/event counts. The individual instruction descriptions describe how to use them.

Table A-5 Atomic Update CPU Load/Store Instructions

Mnemonic	Description	Defined in
LL	Load Linked Word	MIPS II
SC	Store Conditional Word	II
LLD	Load Linked Doubleword	III
SCD	Store Conditional Doubleword	III

A 2.1.4 Coprocessor Loads and Stores

These loads and stores are coprocessor instructions, however it seems more useful to summarize all load and store instructions in one place instead of listing them in the coprocessor instructions functional group.

If a particular coprocessor is not enabled, loads and stores to that processor cannot execute and will cause a Coprocessor Unusable exception. Enabling a coprocessor is a privileged operation provided by the System Control Coprocessor.

Table A-6 Coprocessor Load/Store Instructions

Mnemonic	Description	Defined in
LWCz	Load Word to Coprocessor-z	MIPS I
SWCz	Store Word from Coprocessor-z	I
LDCz	Load Doubleword to Coprocessor-z	II
SDCz	Store Doubleword from Coprocessor-z	II

Table A-7 FPU Load/Store Instructions Using Register + Register Addressing

Mnemonic	Description	Defined in
LWXC1	Load Word Indexed to Floating Point	MIPS IV
SWXC1	Store Word Indexed from Floating Point	IV
LDXC1	Load Doubleword Indexed to Floating Point	IV
SDXC1	Store Doubleword Indexed from Floating Point	IV

A 2.2 Computational Instructions

Two's complement arithmetic is performed on integers represented in two's complement notation. There are signed versions of add, subtract, multiply, and divide. There are add and subtract operations, called "unsigned", that are actually modulo arithmetic without overflow detection. There are unsigned versions of multiply and divide. There is a full complement of shift and logical operations.

MIPS I provides 32-bit integers and 32-bit arithmetic. MIPS III adds 64-bit integers and provides separate arithmetic and shift instructions for 64-bit operands. Logical operations are not sensitive to the width of the register.

A 2.2.5 ALU

Some arithmetic and logical instructions operate on one operand from a register and the other from a 16-bit immediate value in the instruction word. The immediate operand is treated as signed for the arithmetic and compare instructions, and treated as logical (zero-extended to register length) for the logical instructions.

Table A-8 ALU Instructions With an Immediate Operand

Mnemonic	Description	Defined in
ADDI	Add Immediate Word	MIPS I
ADDIU	Add Immediate Unsigned Word	I
SLTI	Set on Less Than Immediate	I
SLTIU	Set on Less Than Immediate Unsigned	I
ANDI	And Immediate	I
ORI	Or Immediate	I
XORI	Exclusive Or Immediate	I
LUI	Load Upper Immediate	I
DADDI	Doubleword Add Immediate	III
DADDIU	Doubleword Add Immediate Unsigned	III

Table A-9 3-Operand ALU Instructions

Mnemonic	Description	Defined in
ADD	Add Word	MIPS I
ADDU	Add Unsigned Word	I
SUB	Subtract Word	I
SUBU	Subtract Unsigned Word	I
DADD	Doubleword Add	III
DADDU	Doubleword Add Unsigned	III
DSUB	Doubleword Subtract	III
DSUBU	Doubleword Subtract Unsigned	III
SLT	Set on Less Than	I
SLTU	Set on Less Than Unsigned	I
AND	And	I
OR	Or	I
XOR	Exclusive Or	I
NOR	Nor	I

A 2.2.6 Shifts

There are shift instructions that take the shift amount from a 5-bit field in the instruction word and shift instructions that take a shift amount from the low-order bits of a general register. The instructions with a fixed shift amount are limited to a 5-bit shift count, so there are separate instructions for doubleword shifts of 0-31 bits and 32-63 bits.

Table A-10 Shift Instructions

Mnemonic	Description	Defined in
SLL	Shift Word Left Logical	MIPS I
SRL	Shift Word Right Logical	I
SRA	Shift Word Right Arithmetic	I
SLLV	Shift Word Left Logical Variable	I
SRLV	Shift Word Right Logical Variable	I
SRAV	Shift Word Right Arithmetic Variable	I
DSLL	Doubleword Shift Left Logical	III
DSRL	Doubleword Shift Right Logical	III
DSRA	Doubleword Shift Right Arithmetic	III
DSLL32	Doubleword Shift Left Logical + 32	III
DSRL32	Doubleword Shift Right Logical + 32	III
DSRA32	Doubleword Shift Right Arithmetic + 32	III
DSLLV	Doubleword Shift Left Logical Variable	III
DSRLV	Doubleword Shift Right Logical Variable	III
DSRAV	Doubleword Shift Right Arithmetic Variable	III

A 2.2.7 Multiply and Divide

The multiply and divide instructions produce twice as many result bits as is typical with other processors and they deliver their results into the HI and LO special registers. Multiply produces a full-width product twice the width of the input operands; the low half is put in LO and the high half is put in HI. Divide produces both a quotient in LO and a remainder in HI. The results are accessed by instructions that transfer data between HI/LO and the general registers.

Table A-11 Multiply/Divide Instructions

Mnemonic	Description	Defined in
MULT	Multiply Word	MIPS I
MULTU	Multiply Unsigned Word	I
DIV	Divide Word	I
DIVU	Divide Unsigned Word	I
DMULT	Doubleword Multiply	III
DMULTU	Doubleword Multiply Unsigned	III
DDIV	Doubleword Divide	III
DDIVU	Doubleword Divide Unsigned	III
MFHI	Move From HI	I
MTHI	Move To HI	I
MFLO	Move From LO	I
MTLO	Move To LO	I

A 2.3 Jump and Branch Instructions

The architecture defines PC-relative conditional branches, a PC-region unconditional jump, an absolute (register) unconditional jump, and a similar set of procedure calls that record a return link address in a general register. For convenience this discussion refers to them all as branches.

All branches have an architectural delay of one instruction. When a branch is taken, the instruction immediately following the branch instruction, in the branch delay slot, is executed before the branch to the target instruction takes place. Conditional branches come in two versions that treat the instruction in the delay slot differently when the branch is not taken and execution falls through. The "branch" instructions execute the instruction in the delay slot, but the "branch likely" instructions do not (they are said to nullify it).

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

Table A-12 Jump Instructions Jumping Within a 256 Megabyte Region

Mnemonic	Description	Defined in
J	Jump	MIPS I
JAL	Jump and Link	I

Table A-13 Jump Instructions to Absolute Address

Mnemonic	Description	Defined in
JR	Jump Register	MIPS I
JALR	Jump and Link Register	I

Table A-14 PC-Relative Conditional Branch Instructions Comparing 2 Registers

Mnemonic	Description	Defined in
BEQ	Branch on Equal	MIPS I
BNE	Branch on Not Equal	I
BLEZ	Branch on Less Than or Equal to Zero	I
BGTZ	Branch on Greater Than Zero	I
BEQL	Branch on Equal Likely	II
BNEL	Branch on Not Equal Likely	II
BLEZL	Branch on Less Than or Equal to Zero Likely	II
BGTZL	Branch on Greater Than Zero Likely	II

Table A-15 PC-Relative Conditional Branch Instructions Comparing Against Zero

Mnemonic	Description	Defined in
BLTZ	Branch on Less Than Zero	MIPS I
BGEZ	Branch on Greater Than or Equal to Zero	I
BLTZAL	Branch on Less Than Zero and Link	I
BGEZAL	Branch on Greater Than or Equal to Zero and Link	I
BLTZL	Branch on Less Than Zero Likely	II
BGEZL	Branch on Greater Than or Equal to Zero Likely	II
BLTZALL	Branch on Less Than Zero and Link Likely	II
BGEZALL	Branch on Greater Than or Equal to Zero and Link Likely	II

A 2.4 Miscellaneous Instructions

A 2.4.1 Exception Instructions

Exception instructions have as their sole purpose causing an exception that will transfer control to a software exception handler in the kernel. System call and breakpoint instructions cause exceptions unconditionally. The trap instructions cause exceptions conditionally based upon the result of a comparison.

Table A-16 System Call and Breakpoint Instructions

Mnemonic	Description	Defined in
SYSCALL	System Call	MIPS I
BREAK	Breakpoint	I

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

Mnemonic	Description	Defined in
TGE	Trap if Greater Than or Equal	MIPS II
TGEU	Trap if Greater Than or Equal Unsigned	II
TLT	Trap if Less Than	II
TLTU	Trap if Less Than Unsigned	II
TEQ	Trap if Equal	II
TNE	Trap if Not Equal	II

Table A-18 Trap-on-Condition Instructions Comparing an Immediate

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

A 2.4.2 Serialization Instructions

The order in which memory accesses from load and store instruction appear outside the processor executing them, in a multiprocessor system for example, is not specified by the architecture. The SYNC instruction creates a point in the executing instruction stream at which the relative order of some loads and stores is known. Loads and stores executed before the SYNC are completed before loads and stores after the SYNC can start.

Table A-19 Serialization Instructions

Mnemonic	Description	Defined in
SYNC	Synchronize Shared Memory	MIPS II

A 2.4.3 Conditional Move Instructions

Instructions were added in MIPS IV to conditionally move one CPU general register to another based on the value in a third general register.

Table A-20 CPU Conditional Move Instructions

Mnemonic	Description	Defined in
MOVN	Move Conditional on Not Zero	MIPS IV
MOVZ	Move Conditional on Zero	IV

A 2.4.4 Prefetch

There are two prefetch advisory instructions; one with register+offset addressing and the other with register+register addressing. These instructions advise that memory is likely to be used in a particular way in the near future and should be prefetched into the cache. The PREFX instruction using register+register addressing mode is coded in the FPU opcode space along with the other operations using register+register addressing.

Table A-21 Prefetch Using Register + Offset Address Mode

Mnemonic	Description	Defined in
PREF	Prefetch Indexed	MIPS IV

Table A-22 Prefetch Using Register + Register Address Mode

Mnemonic	Description	Defined in
PREFX	Prefetch Indexed	MIPS IV

A 2.5 Coprocessor Instructions

Coprocessors are alternate execution units, with register files separate from the CPU. The MIPS architecture provides an abstraction for up to 4 coprocessor units, numbered 0 to 3. Each architecture level defines some of these coprocessors as shown in Table A-23. Coprocessor 0 is always used for system control and coprocessor 1 is used for the floating-point unit. Other coprocessors are architecturally valid, but do not have a reserved use. Some coprocessors are not defined and their opcodes are either reserved or used for other purposes.

Table A-23 Coprocessor Definition and Use in the MIPS Architecture

	MIPS architecture level			
coprocessor	I	II	III	IV
0	Sys Control	Sys Control	Sys Control	Sys Control
1	FPU	FPU	FPU	FPU
2	unused	unused	unused	unused
3	unused	unused	not defined	FPU (COP 1X)

The coprocessors may have two register sets, coprocessor general registers and coprocessor control registers, each set containing up to thirty two registers. Coprocessor computational instructions may alter registers in either set.

System control for all MIPS processors is implemented as coprocessor 0 (CP0), the System Control Coprocessor. It provides the processor control, memory management, and exception handling functions. The CP0 instructions are specific to each CPU and are documented with the CPU-specific information.

If a system includes a floating-point unit, it is implemented as coprocessor 1 (CP1). In MIPS IV, the FPU also uses the computation opcode space for coprocessor unit 3, renamed COP1X. The FPU instructions are documented in Appendix B.

The coprocessor instructions are divided into two main groups:

- Load and store instructions that are reserved in the main opcode space.
- Coprocessor-specific operations that are defined entirely by the coprocessor.

A 2.5.1 Coprocessor Load and Store

Load and store instructions are not defined for CP0; the move to/from coprocessor instructions are the only way to write and read the CP0 registers.

The loads and stores for coprocessors are summarized in Load and Store Instructions on page A-2.

A 2.5.2 Coprocessor Operations

There are up to four coprocessors and the instructions are shown generically for coprocessor-*z*. Within the operation main opcode, the coprocessor has further coprocessor-specific instructions encoded.

Table A-24 Coprocessor Operation Instructions

Mnemonic	Description	Defined in
COPz	Coprocessor-z Operation	MIPS I

A 3 Memory Access Types

MIPS systems provide a few *memory access types* that are characteristic ways to use physical memory and caches to perform a memory access. The memory access type is specified as a cache coherence algorithm (CCA) in the TLB entry for a 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. Implementations without multiprocessor (MP) support provide uncached and cached accesses. Implementations with MP support provide uncached, cached noncoherent and cached coherent accesses. The memory access types use the memory hierarchy as follows:

Uncached

Physical memory is used to resolve the access. Each reference causes a read or write to physical memory. Caches are neither examined nor modified.

Cached Noncoherent

Physical memory and the caches of the processor performing the access are used to resolve the access. Other caches are neither examined nor modified.

Cached Coherent

Physical memory and all caches in the system containing a coherent copy of the physical location are used to resolve the access. A copy of a location is coherent (noncoherent) if the copy was placed in the cache by a cached coherent (cached noncoherent) access. Caches containing a coherent copy of the location are examined and/or modified to keep the contents of the location coherent. It is unpredictable whether caches holding a noncoherent copy of the location are examined and/or modified during a cached coherent access.

Cached

For early 32-bit processors without MP support, cached is equivalent to cached noncoherent. If an instruction description mentions the cached noncoherent access type, the comment applies equally to the cached access type in a processor that has the cached access type.

For processors with MP support, cached is a collective term, e.g. "cached memory" or "cached access", that includes both cached noncoherent and cached coherent. Such a collective use does not imply that cached is an access type, it means that the statement applies equally to cached noncoherent and cached coherent access types.

A 3.1 Mixing References with Different Access Types

It is possible to have more than one virtual location simultaneously mapped to the same physical location. 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.

A processor executing load and store instructions must observe the effect of the load and store instructions to a physical location in the order that they occur in the instruction stream (i.e. program order) for all accesses to virtual locations with the same memory access type.

If a processor executes a load or store using one access type to a physical location, the behavior of a subsequent load or store to the same location using a different memory access type is undefined unless a privileged instruction sequence is executed between the two accesses. Each implementation has a privileged implementation-specific mechanism that must be used to change the access type being used to access a location.

The memory access type of a location affects the behavior of I-fetch, load, store, and prefetch operations to the 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. SYNC affects only load and stores made to locations with uncached or cached coherent memory access types.

A 3.2 Cache Coherence Algorithms and Access Types

The memory access types are specified by implementation-specific cache coherence algorithms (CCAs) in TLB entries. Slightly different cache coherence algorithms such as “cached coherent, update on write” and “cached coherent, exclusive on write” can map to the same memory access type, in this case they both map to cached coherent. In order to map to the same access type the fundamental mechanism of both CCAs must be the same. When it affects the operation of the instruction, the instructions are described in terms of the memory access types. The load and store operations in a processor proceeds according to the specific CCA of the reference, however, and the pseudocode for load and store common functions in the section Load and Store Memory Functions on page A-21 use the CCA value rather than the corresponding memory access type.

A 3.3 Implementation-Specific Access Types

An implementation may provide memory access types other than uncached, cached noncoherent, or cached coherent. Implementation-specific documentation will define the properties of the new access types and their effect on all memory-related operations.

A 4 Description of an Instruction

The CPU instructions are described in alphabetic order. Each description contains several sections that contain specific information about the instruction. The content of the section is described in detail below. An example description is shown in Figure A-1.

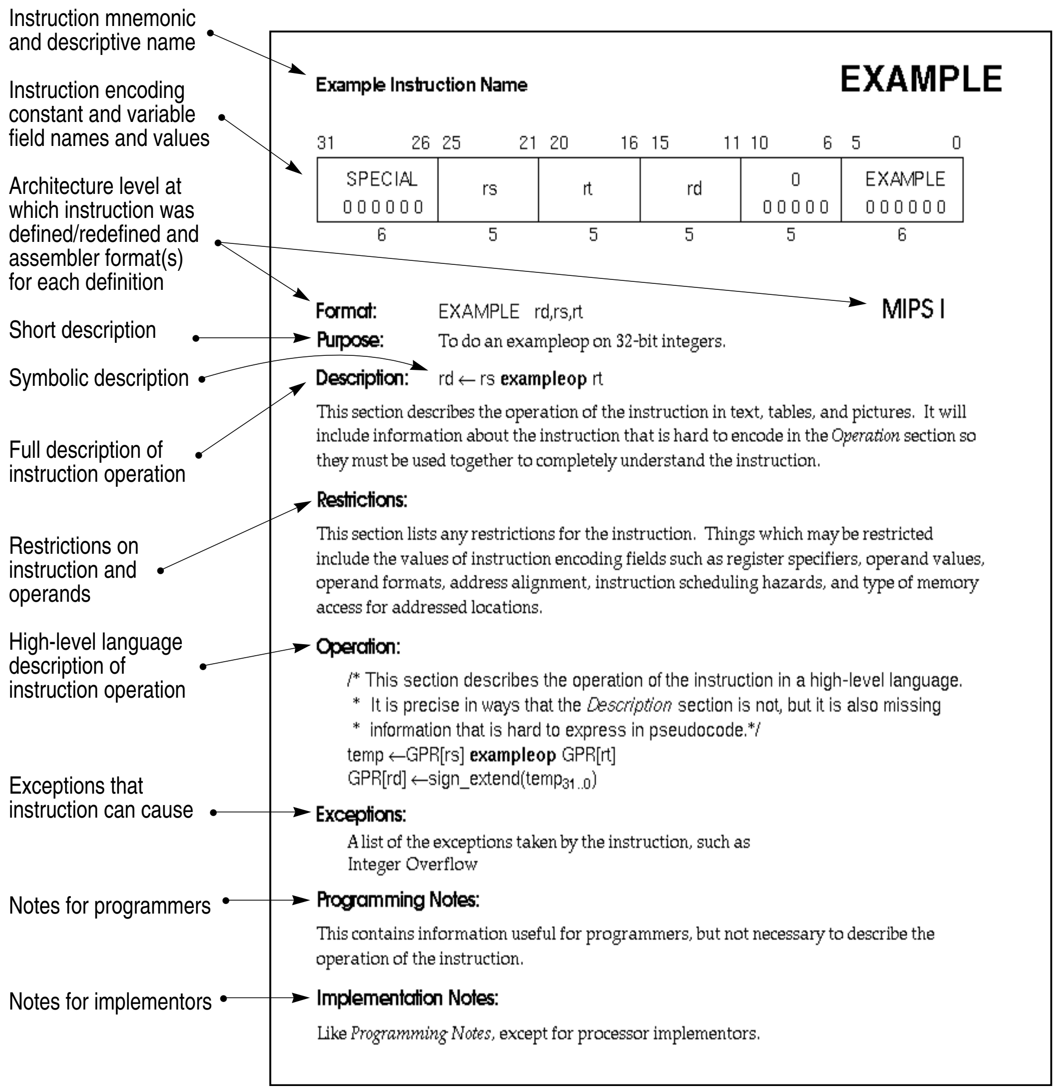


Figure A-1 Example Instruction Description

A 4.1 Instruction mnemonic and name

The instruction mnemonic and name are printed as page headings for each page in the instruction description.

A 4.2 Instruction encoding picture

The instruction word encoding is shown in pictorial form at the top of the instruction description. This picture shows the values of all constant fields and the opcode names for opcode fields in upper-case. It labels all variable fields with lower-case names that are used in the instruction description. Fields that contain zeroes but are not named are unused fields that are required to be zero. A summary of the instruction formats and a definition of the terms used to describe the contents can be found in CPU Instruction Formats on page A-174.

A 4.3 Format

The assembler formats for the instruction and the architecture level at which the instruction was originally defined are shown. If the instruction definition was later extended, the architecture levels at which it was extended and the assembler formats for the extended definition are shown in order of extension. The MIPS architecture levels are inclusive; higher architecture levels include all instructions in previous levels. Extensions to instructions are backwards compatible. The original assembler formats are valid for the extended architecture.

The assembler format is shown with literal parts of the assembler instruction in upper-case characters. The variable parts, the operands, are shown as the lower-case names of the appropriate fields in the instruction encoding picture. The architecture level at which the instruction was first defined, e.g. "MIPS I", is shown at the right side of the page.

There can be more than one assembler format per architecture level. This is sometimes an alternate form of the instruction. Floating-point operations on formatted data show an assembly format with the actual assembler mnemonic for each valid value of the "fmt" field. For example the ADD.fmt instruction shows ADD.S and ADD.D.

The assembler format lines sometimes have comments to the right in parentheses to help explain variations in the formats. The comments are not a part of the assembler format.

A 4.4 Purpose

This is a very short statement of the purpose of the instruction.

A 4.5 Description

If a one-line symbolic description of the instruction is feasible, it will appear immediately to the right of the *Description* heading. The main purpose is to show how fields in the instruction are used in the arithmetic or logical operation.

The body of the section is a description of the operation of the instruction in text, tables, and figures. This description complements the high-level language description in the *Operation* section.

This section uses acronyms for register descriptions. "GPR *rt*" is CPU General Purpose Register specified by the instruction field *rt*. "FPR *fs*" is the Floating Point Operand Register specified by the instruction field *fs*. "CP1 register *fd*" is the coprocessor 1 General Register specified by the instruction field *fd*. "FCSR" is the floating-point control and status register.

A 4.6 Restrictions

This section documents the restrictions on the instruction. Most restrictions fall into one of six categories:

- The valid values for instruction fields (see floating-point ADD.fmt).
- The alignment requirements for memory addresses (see LW).
- The valid values of operands (see DADD).
- The valid operand formats (see floating-point ADD.fmt).
- The order of instructions necessary to guarantee correct execution. These ordering constraints avoid pipeline hazards for which some processors do not have hardware interlocks (see MUL).
- The valid memory access types (see LL/SC).

A 4.7 Operation

This section describes the operation of the instruction as pseudocode in a high-level language notation resembling Pascal. The purpose of this section is to describe the operation of the instruction clearly in a form with less ambiguity than prose. This formal description complements the *Description* section; it is not complete in itself because many of the restrictions are either difficult to include in the pseudocode or omitted for readability.

There will be separate *Operation* sections for 32-bit and 64-bit processors if the operation is different. This is usually necessary because the path to memory is a different size on these processors.

See Operation Section Notation and Functions on page A-18 for more information on the formal notation.

A 4.8 Exceptions

This section lists the exceptions that can be caused by operation of the instruction. It omits exceptions that can be caused by instruction fetch, e.g. TLB Refill. It omits exceptions that can be caused by asynchronous external events, e.g. Interrupt. Although the Bus Error exception may be caused by the operation of a load or store instruction this section does not list Bus Error for load and store instructions because the relationship between load and store instructions and external error indications, like Bus Error, are implementation dependent.

Reserved Instruction is listed for every instruction not in MIPS I because the instruction will cause this exception on a MIPS I processor. To execute a MIPS II, MIPS III, or MIPS IV instruction, the processor must both support the architecture level and have it enabled. The mechanism to do this is implementation specific.

The mechanism used to signal a floating-point unit (FPU) exception is implementation specific. Some implementations use the exception named "Floating Point". Others use external interrupts (the Interrupt exception). This section lists Floating Point to represent all such mechanisms. The specific FPU traps possible are listed, indented, under the Floating Point entry.

The usual floating-point exception model for MIPS architecture processors is precise exceptions. However, the R8000 processor, the first implementation of the MIPS IV architecture, normally operates with imprecise floating-point exceptions. It also has a mode in which it operates with degraded floating-point performance but provides precise exceptions compatible with other MIPS processors. This is mentioned in the description of some floating-point instructions. A general description of this exception model is not included in this document. See the "MIPS R8000 Microprocessor Chip Set Users Manual" for more information.

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

A 4.9 Programming Notes, Implementation Notes

These sections contain material that is useful for programmers and implementors respectively but that is not necessary to describe the instruction and does not belong in the description sections.

A 5 Operation Section Notation and Functions

In an instruction description, the *Operation* section describes the operation performed by each instruction using a high-level language notation. The contents of the *Operation* section are described here. The special symbols and functions used are documented here.

A 5.1 Pseudocode Language

Each of the high-level language statements is executed in sequential order (as modified by conditional and loop constructs).

A 5.2 Pseudocode Symbols

Special symbols used in the notation are described in Table A-25.

Table A-25 Symbols in Instruction Operation Statements

Symbol	Meaning
$\leftarrow$	Assignment.
$=, \neq$	Tests for equality and inequality.
||	Bit string concatenation.
x^y	A y-bit string formed by y copies of the single-bit value x.
$x_{y..z}$	Selection of bits y through z of bit string x. Little-endian bit notation (rightmost bit is 0) is used. If y is less than z, this expression is an empty (zero length) bit string.
+, -	2's complement or floating-point arithmetic: addition, subtraction.
*, $\times$	2's complement or floating-point multiplication (both used for either).
div	2's complement integer division.
mod	2's complement modulo.
/	Floating-point division.
<	2's complement less than comparison.
nor	Bit-wise logical NOR.
xor	Bit-wise logical XOR.
and	Bit-wise logical AND.
or	Bit-wise logical OR.
GPRLEN	The length in bits (32 or 64), of the CPU General Purpose Registers.
GPR[x]	CPU General Purpose Register x. The content of GPR[0] is always zero.
FPR[x]	Floating-Point operand register x.
FCC[cc]	Floating-Point condition code cc. FCC[0] has the same value as COC[1].
FGR[x]	Floating-Point (Coprocessor unit1), general register x.
CPR[z,x]	Coprocessor unit z, general register x.
CCR[z,x]	Coprocessor unit z, control register x.
COC[z]	Coprocessor unit z condition signal.
BigEndianMem	Endian mode as configured at chip reset ($0 \rightarrow$Little, $1 \rightarrow$ Big). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory), and the endianness of Kernel and Supervisor mode execution.
ReverseEndian	Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is effected by setting the RE bit of the Status register. Thus, ReverseEndian may be computed as (SR_{RE} and User mode).
BigEndianCPU	The endianness for load and store instructions ($0 \rightarrow$ Little, $1 \rightarrow$ Big). 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).
LLbit	Bit of virtual state used to specify operation for instructions that provide atomic read-modify-write. It is set when a linked load occurs. It is tested and cleared by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.

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 effects of the pseudocode lines appears to occur (i.e. when the pseudocode is "executed"). 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 – during the instruction time of another instruction. When that happens, the instruction operation is written in sections labelled 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 will have the portion of the instruction operation description that writes the result register in a section labelled "I **+1**:". The effect of pseudocode statements for the current instruction labelled "I **+1**:"appears to occur "at the same time" as the effect of pseudocode statements labelled "I :" for the following instruction. Within one pseudocode sequence the effects of the statements takes place in order. However, between sequences of statements for different instructions that occur "at the same time", there is no order defined. 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 4 before the next instruction time. A taken branch assigns the target address to PC during the instruction time of the instruction in the branch delay slot.
PSIZE	The SIZE, number of bits, of Physical address in an implementation.

A 5.3 Pseudocode Functions

There are several functions used in the pseudocode descriptions. These are used either to make the pseudocode more readable, to abstract implementation specific behavior, or both. The functions are defined in this section.

A 5.3.1 Coprocessor General Register Access Functions

Defined coprocessors, except for CP0, have instructions to exchange words and doublewords between coprocessor general registers and the rest of the system. What a coprocessor does with a word or doubleword supplied to it and how a coprocessor supplies a word or doubleword is defined by the coprocessor itself. This behavior is abstracted into functions:

Table A-26 Coprocessor General Register Access Functions

COP_LW (z, rt, memword) z: The coprocessor unit number. rt: Coprocessor general register specifier. memword: A 32-bit word value supplied to the coprocessor. This is the action taken by coprocessor *z* when supplied with a word from memory during a load word operation. The action is coprocessor specific. The typical action would be to store the contents of *memword* in coprocessor general register *rt*.
COP_LD (z, rt, memdouble) z: The coprocessor unit number. rt: Coprocessor general register specifier. memdouble: 64-bit doubleword value supplied to the coprocessor. This is the action taken by coprocessor *z* when supplied with a doubleword from memory during a load doubleword operation. The action is coprocessor specific. The typical action would be to store the contents of *memdouble* in coprocessor general register *rt*.
dataword ← COP_SW (z, rt) z: The coprocessor unit number. rt: Coprocessor general register specifier. dataword: 32-bit word value. This defines the action taken by coprocessor *z* to supply a word of data during a store word operation. The action is coprocessor specific. The typical action would be to supply the contents of the low-order word in coprocessor general register *rt*.
datadouble ← COP_SD (z, rt) z: The coprocessor unit number. rt: Coprocessor general register specifier. datadouble: 64-bit doubleword value. This defines the action taken by coprocessor *z* to supply a doubleword of data during a store doubleword operation. The action is coprocessor specific. The typical action would be to supply the contents of the doubleword in coprocessor general register *rt*.

A 5.3.2 Load and Store Memory Functions

Regardless of byte ordering (big- or little-endian), the address of a halfword, word, or doubleword is the smallest byte address 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.

In the operation description pseudocode for load and store operations, the functions shown below are used to summarize the handling of virtual addresses and accessing physical memory. The size of the data item to be loaded or stored is passed in the *AccessLength* field. The valid constant names and values are shown in Table A-27. The bytes within the addressed unit of memory (word for 32-bit processors or doubleword for 64-bit processors) which are used can be determined directly from the AccessLength and the two or three low-order bits of the address.

(pAddr, CCA) ←AddressTranslation (vAddr, IorD, LorS)

pAddr:	Physical Address.
CCA:	Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.
vAddr:	Virtual Address.
IorD:	Indicates whether access is for INSTRUCTION or DATA.
LorS:	Indicates whether access is for LOAD or STORE.

Translate a virtual address to a physical address and a cache coherence algorithm describing the mechanism used to resolve the memory reference. Given the virtual address *vAddr*, and whether the reference is to Instructions or Data (*IorD*), find the corresponding physical address (*pAddr*) and the cache coherence algorithm (*CCA*) used to resolve the reference. If the virtual address is in one of the unmapped address spaces the physical address and *CCA* are determined directly by the virtual address. If the virtual address is in one of the mapped address spaces then the TLB is used to determine the physical address and access type; if the required translation is not present in the TLB or the desired access is not permitted the function fails and an exception is taken.

MemElem ← LoadMemory (CCA, AccessLength, pAddr, vAddr, IorD)

MemElem:	Data is returned in a fixed width with a natural alignment. The width is the same size as the CPU general purpose register, 32 or 64 bits, aligned on a 32 or 64-bit boundary respectively.
CCA:	Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.
AccessLength:	Length, in bytes, of access.
pAddr:	Physical Address.
vAddr:	Virtual Address.
IorD:	Indicates whether access is for Instructions or Data.

Load a value from memory.
Uses the cache and main memory as specified in the Cache Coherence Algorithm (*CCA*) and the sort of access (*IorD*) to find the contents of *AccessLength* memory bytes starting at physical location *pAddr*. The data is returned in the fixed width naturally-aligned memory element (MemElem). The low-order two (or three) bits of the address and the *AccessLength* indicate which of the bytes within *MemElem* needs to be given to the processor. If the memory access type of the reference is uncached then only the referenced bytes are read from memory and valid within the memory element. If the access type is cached, and the data is not present in cache, an implementation specific size and alignment block of memory is read and loaded into the cache to satisfy a load reference. At a minimum, the block is the entire memory element.

StoreMemory (CCA, AccessLength, MemElem, pAddr, vAddr)

CCA:	Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.
AccessLength:	Length, in bytes, of access.
MemElem:	Data in the width and alignment of a memory element. The width is the same size as the CPU general purpose register, 4 or 8 bytes, aligned on a 4 or 8-byte boundary. For a partial-memory-element store, only the bytes that will be stored must be valid.
pAddr:	Physical Address.
vAddr:	Virtual Address.

Store a value to memory.
The specified data is stored into the physical location *pAddr* using the memory hierarchy (data caches and main memory) as specified by the Cache Coherence Algorithm (*CCA*). The *MemElem* contains the data for an aligned, fixed-width memory element (word for 32-bit processors, doubleword for 64-bit processors), though only the bytes that will actually be stored to memory need to be valid. The low-order two (or three) bits of *pAddr* and the *AccessLength* field indicates which of the bytes within the *MemElem* data should actually be stored; only these bytes in memory will be changed.

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

CCA:	Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.
pAddr:	physical Address.
vAddr:	Virtual Address.
DATA:	Indicates that access is for DATA.
hint:	hint that indicates the possible use of the data.

Prefetch data from memory.
Prefetch is an advisory instruction for which an implementation specific action is taken. The action taken may increase performance but must not change the meaning of the program or alter architecturally-visible state.

Table A-27 AccessLength Specifications for Loads/Stores

AccessLength Name	Value	Meaning
DOUBLEWORD	7	8 bytes (64 bits)
SEPTIBYTE	6	7 bytes (56 bits)
SEXTIBYTE	5	6 bytes (48 bits)
QUINTIBYTE	4	5 bytes (40 bits)
WORD	3	4 bytes (32 bits)
TRIPLEBYTE	2	3 bytes (24 bits)
HALFWORD	1	2 bytes (16 bits)
BYTE	0	1 byte (8 bits)

A 5.3.3 Access Functions for Floating-Point Registers

The details of the relationship between CP1 general registers and floating-point operand registers is encapsulated in the functions included in this section. See Valid Operands for FP Instructions on page B-24 for more information.

This function returns the current logical width, in bits, of the CP1 general registers. All 32-bit processors will return "32". 64-bit processors will return "32" when in 32-bit-CP1-register emulation mode and "64" when in native 64-bit mode.

The following pseudocode referring to the $Status_{FR}$ bit is valid for all existing MIPS 64-bit processors at the time of this writing, however this is a privileged processor-specific mechanism and it may be different in some future processor.

```
SizeFGR()   --  current size, in bits, of the CP1 general registers
size ←SizeFGR()
    if 32_bit_processor then
        size ← 32
    else
        /* 64-bit processor */
        if StatusFR = 1 then
            size ← 64
        else
            size ← 32
        endif
    endif
```

This pseudocode specifies how the unformatted contents loaded or moved-to CP1 registers are interpreted to form a formatted value. If an FPR contains a value in some format, rather than unformatted contents from a load (uninterpreted), it is valid to interpret the value in that format, but not to interpret it in a different format.

```
ValueFPR()  --  Get a formatted value from an FPR.
value ←ValueFPR (fpr, fmt)      /* get a formatted value from an FPR */
    if SizeFGR() = 64 then
        case fmt of
            S, W:
                value ← FGR[fpr]31..0
            D, L:
                value ← FGR[fpr]
        endcase
    elseif fpr0 = 0 then            /* fpr is valid (even), 32-bit wide FGRs */
        case fmt of
            S, W:
                value ← FGR[fpr]
            D, L:
                value ← FGR[fpr+1] || FGR[fpr]
        endcase
    else                            /* undefined for odd 32-bit FGRs */
        UndefinedResult
    endif
```

This pseudocode specifies the way that a binary encoding representing a formatted value is stored into CP1 registers by a computational or move operation. This binary representation is visible to store or move-from instructions. Once an FPR contains a value via StoreFPR(), it is not valid to interpret the value with ValueFPR() in a different format.

```
StoreFPR()  --  store a formatted value into an FPR.
StoreFPR(fpr, fmt, value):          /* place a formatted value into an FPR */
    if SizeFGR() = 64 then   /* 64-bit wide FGRs */
        case fmt of
            S, W:
                FGR[fpr] ← undefined^32 || value
            D, L:
                FGR[fpr] ← value
        endcase
    elseif fpr_0 = 0 then               /* fpr is valid (even), 32-bit wide FGRs */
        case fmt of
            S, W:
                FGR[fpr+1] ← undefined^32
                FGR[fpr] ← value
            D, L:
                FGR[fpr+1] ← value_63..32
                FGR[fpr] ← value_31..0
        endcase
    else                                /* undefined for odd 32-bit FGRs */
        UndefinedResult
    endif
```

A 5.3.4 Miscellaneous Functions

SyncOperation(stype) stype: Type of load/store ordering to perform. order loads and stores to synchronize shared memory. Perform the action necessary to make the effects of groups synchronizable loads and stores indicated by *stype* occur in the same order for all processors.
SignalException(Exception) Exception The exception condition that exists. Signal an exception condition. This will result in an exception that aborts the instruction. The instruction operation pseudocode will never see a return from this function call.
UndefinedResult() This function indicates that the result of the operation is undefined.

NullifyCurrentInstruction() Nullify the current instruction. This occurs during the instruction time for some instruction and that instruction is not executed further. This appears for branch-likely instructions during the execution of the instruction in the delay slot and it kills the instruction in the delay slot.
CoprocessorOperation (z, cop_fun) z Coprocessor unit number cop_fun Coprocessor function from function field of instruction Perform the specified Coprocessor operation.

A 6 Individual CPU Instruction Descriptions

The user-mode CPU instructions are described in alphabetic order. See Description of an Instruction on page A-15 for a description of the information in each instruction description.

ADD

Add Word

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	ADD 1 0 0 0 0 0
6	5	5	5	5	6

Format: ADD rd, rs, rt **MIPS I**

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

Description: rd ← rs + rt

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
temp ←GPR[rs] + GPR[rt]
if (32_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rd] ←sign_extend(temp31..0)
endif
```

Exceptions:

Integer Overflow

Programming Notes:

ADDU performs the same arithmetic operation but, does not trap on overflow.

Add Immediate Word ADDI

31–26	25–21	20–16	15–0
ADDI 0 0 1 0 0 0	rs	rt	immediate
6	5	5	16

Format: ADDI rt, rs, immediate **MIPS I**

Purpose: To add a constant to a 32-bit integer. If overflow occurs, then trap.

Description: rt ← rs + immediate

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rt*.

Restrictions:

On 64-bit processors, if GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue(GPR[rs])) then UndefinedResult() endif
temp ←GPR[rs] + sign_extend(immediate)
if (32_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rt] ←sign_extend(temp31..0)
endif
```

Exceptions:

Integer Overflow

Programming Notes:

ADDIU performs the same arithmetic operation but, does not trap on overflow.

ADDIU

Add Immediate Unsigned Word

31–26	25–21	20–16	15–0
ADDIU 0 0 1 0 0 1	rs	rt	immediate
6	5	5	16

Format: ADDIU rt, rs, immediate **MIPS I**

Purpose: To add a constant to a 32-bit integer.

Description: rt ← rs + immediate

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue(GPR[rs])) then UndefinedResult() endif
temp     ←GPR[rs] + sign_extend(immediate)
GPR[rt] ← sign_extend(temp31..0)
```

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as “C” language arithmetic.

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	ADDU 100001
6	5	5	5	5	6

Format: ADDU rd, rs, rt **MIPS I**

Purpose: To add 32-bit integers.

Description: rd ← rs + rt

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
temp    ←GPR[rs] + GPR[rt]
GPR[rd]← sign_extend(temp31..0)
```

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as “C” language arithmetic.

AND

And

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	AND 1 0 0 1 0 0
6	5	5	5	5	6

Format: AND rd, rs, rt **MIPS I**

Purpose: To do a bitwise logical AND.

Description: rd ← rs AND rt

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical AND operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

GPR[rd] ← GPR[rs] and GPR[rt]

Exceptions:

None

And Immediate ANDI

31–26	25–21	20–16	15–0
ANDI 0 0 1 1 0 0	rs	rt	immediate
6	5	5	16

Format: ANDI rt, rs, immediate **MIPS I**

Purpose: To do a bitwise logical AND with a constant.

Description: rt ← rs AND immediate

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical AND operation. The result is placed into GPR *rt*.

Restrictions:

None

Operation:

GPR[rt] ← zero_extend(immediate) and GPR[rs]

Exceptions:

None

BEQ

Branch on Equal

31–26	25–21	20–16	15–0
BEQ 0 0 0 1 0 0	rs	rt	offset
6	5	5	16

Format: BEQ rs, rt, offset **MIPS I**

Purpose: To compare GPRs then do a PC-relative conditional branch.

Description: if (rs = rt) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend(offset || 0^2)
      condition ← (GPR[rs] = GPR[rt])
I+1:  if condition then
          PC ← PC + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

Branch on Equal Likely **BEQL**

31–26	25–21	20–16	15–0
BEQL 0 1 0 1 0 0	rs	rt	offset
6	5	5	16

Format: BEQL rs, rt, offset **MIPS II**

Purpose: To compare GPRs then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs = rt) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend(offset || 0^2)
      condition ← (GPR[rs] = GPR[rt])
I+1:  if condition then
          PC ← PC + tgt_offset
      else
          NullifyCurrentInstruction()
      endif
```

Exceptions:

Reserved Instruction

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BGEZ

Branch on Greater Than or Equal to Zero

31–26	25–21	20–16	15–0
REGIMM 000001	rs	BGEZ 00001	offset
6	5	5	16

Format: BGEZ rs, offset **MIPS I**

Purpose: To test a GPR then do a PC-relative conditional branch.

Description: if (rs ≥ 0) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend(offset || 0^2)
      condition ← GPR[rs] ≥ 0^GPRLEN
I+1:  if condition then
          PC ← PC + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZAL 10001	offset
6	5	5	16

Format: BGEZAL rs, offset MIPS I

Purpose: To test a GPR then do a PC-relative conditional procedure call.

Description: if (rs ≥ 0) then procedure_call

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

GPR 31 must not be used for the source register *rs*, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

Operation:

```
I:    tgt_offset ← sign_extend(offset || 0^2)
      condition ← GPR[rs] ≥ 0^GPRLEN
      GPR[31] ← PC + 8
I+1:  if condition then
          PC ← PC + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to more distant addresses.

BGEZALL Branch on Greater Than or Equal to Zero and Link Likely

31–26	25–21	20–16	15–0
REGIMM 0 0 0 0 0 1	rs	BGEZALL 1 0 0 1 1	offset
6	5	5	16

Format: BGEZALL rs, offset **MIPS II**

Purpose: To test a GPR then do a PC-relative conditional procedure call; execute the delay slot only if the branch is taken.

Description: if (rs ≥ 0) then procedure_call_likely

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

GPR 31 must not be used for the source register *rs*, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

Operation:

```
I:    tgt_offset ← sign_extend(offset || 0^2)
      condition ← GPR[rs] ≥ 0^GPRLEN
      GPR[31] ← PC + 8
I+1:  if condition then
          PC ← PC + tgt_offset
      else
          NullifyCurrentInstruction()
      endif
```

Exceptions:

Reserved Instruction

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to more distant addresses.

Branch on Greater Than or Equal to Zero Likely **BGEZL**

31–26	25–21	20–16	15–0
REGIMM 000001	rs	BGEZL 00011	offset
6	5	5	16

Format: BGEZL rs, offset **MIPS II**

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs ≥ 0) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operation:

```
I:     tgt_offset ← sign_extend(offset || 0^2)
       condition ← GPR[rs] ≥ 0^GPRLEN
I+1:   if condition then
           PC ← PC + tgt_offset
       else
           NullifyCurrentInstruction()
       endif
```

Exceptions:

Reserved Instruction

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BGTZ

Branch on Greater Than Zero

31–26	25–21	20–16	15–0
BGTZ 0 0 0 1 1 1	rs	0 0 0 0 0 0	offset
6	5	5	16

Format: BGTZ rs, offset **MIPS I**

Purpose: To test a GPR then do a PC-relative conditional branch.

Description: if (rs > 0) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend(offset || 0^2)
      condition ← GPR[rs] > 0^GPRLEN
I+1:  if condition then
          PC ← PC + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

Branch on Greater Than Zero Likely **BGTZL**

31–26	25–21	20–16	15–0
BGTZL 010111	rs	0 00000	offset
6	5	5	16

Format: BGTZL rs, offset **MIPS II**

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs > 0) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operation:

```
I:      tgt_offset ← sign_extend(offset || 0^2)
        condition ← GPR[rs] > 0^GPRLEN
I+1:    if condition then
            PC ← PC + tgt_offset
        else
            NullifyCurrentInstruction()
        endif
```

Exceptions:

Reserved Instruction

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BLEZ

Branch on Less Than or Equal to Zero

31–26	25–21	20–16	15–0
BLEZ 000110	rs	0 00000	offset
6	5	5	16

Format: BLEZ rs, offset **MIPS I**

Purpose: To test a GPR then do a PC-relative conditional branch.

Description: if (rs ≤ 0) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:      tgt_offset ← sign_extend(offset || 0^2)
        condition ← GPR[rs] ≤ 0^GPRLEN
I+ 1 :if condition then
            PC ← PC + tgt_offset

        endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

Branch on Less Than or Equal to Zero Likely BLEZL

31–26	25–21	20–16	15–0
BLEZL 0 1 0 1 1 0	rs	0 0 0 0 0 0	offset
6	5	5	16

Format: BLEZL rs, offset **MIPS II**

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs ≤ 0) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operation:

```
I:      tgt_offset ← sign_extend(offset || 0^2)
        condition ← GPR[rs] ≤ 0^GPRLEN
I+1:    if condition then
            PC ← PC + tgt_offset
        else
            NullifyCurrentInstruction()
        endif
```

Exceptions:

Reserved Instruction

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BLTZ — Branch on Less Than Zero

31–26	25–21	20–16	15–0
REGIMM 000001	rs	BLTZ 00000	offset
6	5	5	16

Format: BLTZ rs, offset — **MIPS I**

Purpose: To test a GPR then do a PC-relative conditional branch.

Description: if (rs < 0) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:     tgt_offset ← sign_extend(offset || 0^2)
       condition ← GPR[rs] < 0^GPRLEN
I+1:   if condition then
           PC ← PC + tgt_offset
       endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

Branch on Less Than Zero And Link **BLTZAL**

31–26	25–21	20–16	15–0
REGIMM 000001	rs	BLTZAL 10000	offset
6	5	5	16

Format: BLTZAL rs, offset **MIPS I**

Purpose: To test a GPR then do a PC-relative conditional procedure call.

Description: if (rs < 0) then procedure_call

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch (not the branch itself), where execution would continue after a procedure call.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch, in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

GPR 31 must not be used for the source register *rs*, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

Operation:

```
I:    tgt_offset ← sign_extend(offset || 0^2)
      condition ← GPR[rs] < 0^GPRLEN
      GPR[31] ← PC + 8
I+1 : if condition then
          PC ← PC + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to more distant addresses.

BLTZALL — Branch on Less Than Zero And Link Likely

31–26	25–21	20–16	15–0
REGIMM 000001	rs	BLTZALL 10010	offset
6	5	5	16

Format: BLTZALL rs, offset — MIPS II

Purpose: To test a GPR then do a PC-relative conditional procedure call; execute the delay slot only if the branch is taken.

Description: if (rs < 0) then procedure_call_likely

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch (not the branch itself), where execution would continue after a procedure call.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch, in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

GPR 31 must not be used for the source register *rs*, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

Operation:

```
I:    tgt_offset ← sign_extend(offset || 0^2)
      condition ← GPR[rs] < 0^GPRLEN
      GPR[31] ← PC + 8
I+1:  if condition then
          PC ← PC + tgt_offset
      else
          NullifyCurrentInstruction()
      endif
```

Exceptions:

Reserved Instruction

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to more distant addresses.

Branch on Less Than Zero Likely **BLTZL**

31–26	25–21	20–16	15–0
REGIMM 000001	rs	BLTZL 00010	offset
6	5	5	16

Format: BLTZ rs, offset **MIPS II**

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs < 0) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operation:

```
I:     tgt_offset ← sign_extend(offset || 0^2)
       condition ← GPR[rs] < 0^GPRLEN
I+1:   if condition then
           PC ← PC + tgt_offset
       else
           NullifyCurrentInstruction()
       endif
```

Exceptions:

Reserved Instruction

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BNE

Branch on Not Equal

31 — 26	25 — 21	20 — 16	15 — 0
BNE 000101	rs	rt	offset
6	5	5	16

Format: BNE rs, rt, offset **MIPS I**

Purpose: To compare GPRs then do a PC-relative conditional branch.

Description: if (rs ≠ rt) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are not equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:     tgt_offset ← sign_extend(offset || 0^2)
       condition ← (GPR[rs] ≠ GPR[rt])
I+1:   if condition then
           PC ← PC + tgt_offset
       endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

31–26	25–21	20–16	15–0
BNEL 0 1 0 1 0 1	rs	rt	offset
6	5	5	16

Format: BNEL rs, rt, offset **MIPS II**

Purpose: To compare GPRs then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs ≠ rt) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are not equal, branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operation:

```
I:     tgt_offset ← sign_extend(offset || 0^2)
       condition ← (GPR[rs] ≠ GPR[rt])
I+ 1 :if condition then
           PC ← PC + tgt_offset
       else
           NullifyCurrentInstruction()
       endif
```

Exceptions:

Reserved Instruction

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BREAK

Breakpoint

31 26	25 6	5 0
SPECIAL 0 0 0 0 0 0	code	BREAK 0 0 1 1 0 1
6	20	6

Format: BREAK **MIPS I**

Purpose: To cause a Breakpoint exception.

Description:

A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Restrictions:

None

Operation:

```
SignalException(Breakpoint)
```

Exceptions:

Breakpoint

Coprocessor Operation COPz

31 — 26	25 — 0
COPz 0 1 0 0 z z	cop_fun
6	26

Format: COP0 cop_fun
COP1 cop_fun
COP2 cop_fun
COP3 cop_fun

MIPS I

Purpose: To execute a coprocessor instruction.

Description:

The coprocessor operation specified by *cop_fun* is performed by coprocessor unit *zz*. Details of coprocessor operations must be found in the specification for each coprocessor.

Each MIPS architecture level defines up to 4 coprocessor units, numbered 0 to 3 (see Coprocessor Instructions on page A-11). The opcodes corresponding to coprocessors that are not defined by an architecture level may be used for other instructions.

Restrictions:

Access to the coprocessors is controlled by system software. Each coprocessor has a "coprocessor usable" bit in the System Control coprocessor. The usable bit must be set for a user program to execute a coprocessor instruction. If the usable bit is not set, an attempt to execute the instruction will result in a Coprocessor Unusable exception. An unimplemented coprocessor must never be enabled. The result of executing this instruction for an unimplemented coprocessor when the usable bit is set, is undefined.

See specification for the specific coprocessor being programmed.

Operation:

```
CoprocessorOperation (z, cop_fun)
```

Exceptions:

Reserved Instruction
Coprocessor Unusable
Coprocessor interrupt or Floating-Point Exception (CP1 only for some processors)

DADD

Doubleword Add

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DADD 1 0 1 1 0 0
6	5	5	5	5	6

Format: DADD rd, rs, rt **MIPS III**

Purpose: To add 64-bit integers. If overflow occurs, then trap.

Description: rd ← rs + rt

The 64-bit doubleword value in GPR *rt* is added to the 64-bit value in GPR *rs* to produce a 64-bit result. If the addition results in 64-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into GPR *rd*.

Restrictions:

None

Operation: 64- bit processors

```
temp ← GPR[rs] + GPR[rt]
if (64_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp
endif
```

Exceptions:

Integer Overflow
Reserved Instruction

Programming Notes:

DADDU performs the same arithmetic operation but, does not trap on overflow.

Doubleword Add Immediate **DADDI**

31 26	25 21	20 16	15 0
DADDI 0 1 1 0 0 0	rs	rt	immediate
6	5	5	16

Format: DADDI rt, rs, immediate **MIPS III**

Purpose: To add a constant to a 64-bit integer. If overflow occurs, then trap.

Description: rt ← rs + immediate

The 16-bit signed *immediate* is added to the 64-bit value in GPR *rs* to produce a 64-bit result. If the addition results in 64-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into GPR *rt*.

Restrictions:

None

Operation: 64- bit processors

```
temp ← GPR[rs] + sign_extend(immediate)
if (64_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rt] ← temp
endif
```

Exceptions:

Integer Overflow
Reserved Instruction

Programming Notes:

DADDIU performs the same arithmetic operation but, does not trap on overflow.

DADDIU

Doubleword Add Immediate Unsigned

31 — 26	25 — 21	20 — 16	15 — 0
DADDIU 0 1 1 0 0 1	rs	rt	immediate
6	5	5	16

Format: DADDIU rt, rs, immediate **MIPS III**

Purpose: To add a constant to a 64-bit integer.

Description: rt ← rs + immediate

The 16-bit signed *immediate* is added to the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation: 64- bit processors

GPR[rt] ← GPR[rs] + sign_extend(immediate)

Exceptions:

Reserved Instruction

Programming Notes:

The term "unsigned" in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as "C" language arithmetic.

Doubleword Add Unsigned DADDU

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DADDU 1 0 1 1 0 1
6	5	5	5	5	6

Format: DADDU rd, rs, rt **MIPS III**

Purpose: To add 64-bit integers.

Description: rd ← rs + rt

The 64-bit doubleword value in GPR *rt* is added to the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation: 64- bit processors

GPR[rd] ←GPR[rs] + GPR[rt]

Exceptions:

Reserved Instruction

Programming Notes:

The term "unsigned" in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as "C" language arithmetic.

DDIV

Doubleword Divide

31–26	25–21	20–16	15–6	5–0
SPECIAL 000000	rs	rt	0 00 0000 0000	DDIV 011110
6	5	5	10	6

Format: DDIV rs, rt **MIPS III**

Purpose: To divide 64-bit signed integers.

Description: (LO, HI) ← rs / rt

The 64-bit doubleword in GPR *rs* is divided by the 64-bit doubleword in GPR *rt*, treating both operands as signed values. The 64-bit quotient is placed into special register *LO* and the 64-bit remainder is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the *HI* or *LO* special registers must be separated from subsequent instructions that write to them by two or more other instructions.

If the divisor in GPR *rt* is zero, the arithmetic result value is undefined.

Operation: 64- bit processors

```
I-2:, I-1:   LO, HI  ← undefined
I:           LO      ← GPR[rs] div GPR[rt]
             HI      ← GPR[rs] mod GPR[rt]
```

Exceptions:

Reserved Instruction

Programming Notes:

See the Programming Notes for the DIV instruction.

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DDIVU 0 1 1 1 1 1
6	5	5	10	6

Format: DDIVU rs, rt **MIPS III**

Purpose: To divide 64-bit unsigned integers.

Description: (LO, HI) ← rs / rt

The 64-bit doubleword in GPR *rs* is divided by the 64-bit doubleword in GPR *rt*, treating both operands as unsigned values. The 64-bit quotient is placed into special register *LO* and the 64-bit remainder is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the *HI* or *LO* special registers must be separated from subsequent instructions that write to them by two or more other instructions.

If the divisor in GPR *rt* is zero, the arithmetic result value is undefined.

Operation: 64- bit processors

```
I-2:, I-1:  LO, HI  ← undefined
I:          LO      ← (0 || GPR[rs]) div (0 || GPR[rt])
            HI      ← (0 || GPR[rs]) mod (0 || GPR[rt])
```

Exceptions:

Reserved instruction

Programming Notes:

See the Programming Notes for the DIV instruction.

DIV

Divide Word

31–26	25–21	20–16	15–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DIV 0 1 1 0 1 0
6	5	5	10	6

Format: DIV rs, rt **MIPS I**

Purpose: To divide 32-bit signed integers.

Description: (LO, HI) ← rs / rt

The 32-bit word value in GPR *rs* is divided by the 32-bit value in GPR *rt*, treating both operands as signed values. The 32-bit quotient is placed into special register *LO* and the 32-bit remainder is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the *HI* or *LO* special registers must be separated from subsequent instructions that write to them by two or more other instructions.

If the divisor in GPR *rt* is zero, the arithmetic result value is undefined.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
I-2:, I-1:  LO, HI  ← undefined
I:          q       ← GPR[rs]31..0 div GPR[rt]31..0
            LO      ← sign_extend(q31..0)
            r       ← GPR[rs]31..0 mod GPR[rt]31..0
            HI      ← sign_extend(r31..0)
```

Exceptions:

None

Programming Notes:

In some processors the integer divide operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written will wait (interlock) until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the divide so that other instructions can execute in parallel.

No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions should be detected and some action taken, then the divide instruction is typically followed by additional instructions to check for a zero divisor and/or for overflow. If the divide is asynchronous then the zero-divisor check can execute in parallel with the divide. The action taken on either divide-by-zero or overflow is either a convention within the program itself or more typically, the system software; one possibility is to take a BREAK exception with a code field value to signal the problem to the system software.

As an example, the C programming language in a UNIX environment expects division by zero to either terminate the program or execute a program-specified signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test for a zero divisor and execute a BREAK instruction to inform the operating system if one is detected.

DIVU

Divide Unsigned Word

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DIVU 0 1 1 0 1 1
6	5	5	10	6

Format: DIVU rs, rt **MIPS I**

Purpose: To divide 32-bit unsigned integers.

Description: (LO, HI) ← rs / rt

The 32-bit word value in GPR *rs* is divided by the 32-bit value in GPR *rt*, treating both operands as unsigned values. The 32-bit quotient is placed into special register *LO* and the 32-bit remainder is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the *HI* or *LO* special registers must be separated from subsequent instructions that write to them, like this one, by two or more other instructions.

If the divisor in GPR *rt* is zero, the arithmetic result is undefined.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
I-2:, I-1:  LO, HI  ← undefined
I:          q       ← (0 || GPR[rs]31..0) div (0 || GPR[rt]31..0)
            LO      ← sign_extend(q31..0)
            r       ← (0 || GPR[rs]31..0) mod (0 || GPR[rt]31..0)
            HI      ← sign_extend(r31..0)
```

Exceptions:

None

Programming Notes:

See the Programming Notes for the DIV instruction.

Doubleword Multiply **DMULT**

31–26	25–21	20–16	15–6	5–0
SPECIAL 000000	rs	rt	0 00 0000 0000	DMULT 011100
6	5	5	10	6

Format: DMULT rs, rt **MIPS III**

Purpose: To multiply 64-bit signed integers.

Description: (LO, HI) $\leftarrow$ rs $\times$ rt

The 64-bit doubleword value in GPR *rt* is multiplied by the 64-bit value in GPR *rs*, treating both operands as signed values, to produce a 128-bit result. The low-order 64-bit doubleword of the result is placed into special register *LO*, and the high-order 64-bit doubleword is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the *HI* or *LO* special registers must be separated from subsequent instructions that write to them by two or more other instructions.

Operation: 64- bit processors

```
I-2:, I-1:LO, HI  ← undefined
I:        prod    ← GPR[rs] * GPR[rt]
          LO      ← prod63..0
          H I     ← prod127..64
```

Exceptions:

Reserved Instruction

Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written will wait (interlock) until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

DMULTU

Doubleword Multiply Unsigned

31–26	25–21	20–16	15–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DMULTU 0 1 1 1 0 1
6	5	5	10	6

Format: DMULTU rs, rt **MIPS III**

Purpose: To multiply 64-bit unsigned integers.

Description: (LO, HI) ← rs × rt

The 64-bit doubleword value in GPR *rt* is multiplied by the 64-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 128-bit result. The low-order 64-bit doubleword of the result is placed into special register *LO*, and the high-order 64-bit doubleword is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the *HI* or *LO* special registers must be separated from subsequent instructions that write to them by two or more other instructions.

Operation: 64- bit processors

```
I-2:, I-1:LO, HI   ← undefined
I:        prod     ← (0 || GPR[rs]) * (0 || GPR[rt])
          LO       ← prod63..0
          HI       ← prod127..64
```

Exceptions:

Reserved Instruction

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSLL 1 1 1 0 0 0
6	5	5	5	5	6

Format: DSLL rd, rt, sa **MIPS III**

Purpose: To left shift a doubleword by a fixed amount — 0 to 31 bits.

Description: rd ← rt << sa

The 64-bit doubleword contents of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 0 to 31 is specified by *sa.*

Restrictions:

None

Operation: 64- bit processors

$s \leftarrow 0 \mid\mid sa$
$GPR[rd] \leftarrow GPR[rt]_{(63-s)..0} \mid\mid 0^{s}$

Exceptions:

Reserved Instruction

DSLL32

Doubleword Shift Left Logical Plus 32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSLL32 1 1 1 1 0 0
6	5	5	5	5	6

Format: DSLL32 rd, rt, sa **MIPS III**

Purpose: To left shift a doubleword by a fixed amount — 32 to 63 bits.

Description: rd ← rt << (sa+32)

The 64-bit doubleword contents of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 32 to 63 is specified by *sa*+32.

Restrictions:

None

Operation: 64- bit processors

s ← 1 || sa /* 32+sa */
$GPR[rd] \leftarrow GPR[rt]_{(63-s)..0} \,||\, 0^{s}$

Exceptions:

Reserved Instruction

Doubleword Shift Left Logical Variable DSLLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DSLLV 0 1 0 1 0 0
6	5	5	5	5	6

Format: DSLLV rd, rt, rs **MIPS III**

Purpose: To left shift a doubleword by a variable number of bits.

Description: rd ← rt << rs

The 64-bit doubleword contents of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 0 to 63 is specified by the low-order six bits in GPR *rs.*

Restrictions:

None

Operation: 64- bit processors

$$s \leftarrow 0 \mathbin{||} GPR[rs]_{5..0}$$
$$GPR[rd] \leftarrow GPR[rt]_{(63-s)..0} \mathbin{||} 0^{s}$$

Exceptions:

Reserved Instruction

DSRA

Doubleword Shift Right Arithmetic

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSRA 1 1 1 0 1 1
6	5	5	5	5	6

Format: DSRA rd, rt, sa **MIPS III**

Purpose: To arithmetic right shift a doubleword by a fixed amount — 0 to 31 bits.

Description: rd ← rt >> sa (arithmetic)

The 64-bit doubleword contents of GPR *rt* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 0 to 31 is specified by *sa.*

Restrictions:

None

Operation: 64- bit processors

$s \leftarrow 0 \,||\, sa$

$GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,||\, GPR[rt]_{63..s}$

Exceptions:

Reserved Instruction

Doubleword Shift Right Arithmetic Plus 32 DSRA32

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSRA32 1 1 1 1 1 1
6	5	5	5	5	6

Format: DSRA32 rd, rt, sa **MIPS III**

Purpose: To arithmetic right shift a doubleword by a fixed amount — 32-63 bits.

Description: rd ← rt >> (sa+32) (arithmetic)

The doubleword contents of GPR *rt* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 32 to 63 is specified by *sa*+32.

Restrictions:

None

Operation: 64- bit processors

```
s          ← 1 || sa          /* 32+sa */
GPR[rd]← (GPR[rt]63)^s || GPR[rt] 63..s
```

$s \leftarrow 1 \,||\, sa$ /* 32+sa */

$GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,||\, GPR[rt]_{63..s}$

Exceptions:

Reserved Instruction

DSRAV

Doubleword Shift Right Arithmetic Variable

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DSRAV 0 1 0 1 1 1
6	5	5	5	5	6

Format: DSRAV rd, rt, rs **MIPS III**

Purpose: To arithmetic right shift a doubleword by a variable number of bits.

Description: rd ← rt >> rs (arithmetic)

The doubleword contents of GPR *rt* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 0 to 63 is specified by the low-order six bits in GPR *rs.*

Restrictions:

None

Operation: 64- bit processors

$s \quad \leftarrow GPR[rs]_{5..0}$
$GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \;||\; GPR[rt]_{63..s}$

Exceptions:

Reserved Instruction

Doubleword Shift Right Logical **DSRL**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSRL 1 1 1 0 1 0
6	5	5	5	5	6

Format: DSRL rd, rt, sa **MIPS III**

Purpose: To logical right shift a doubleword by a fixed amount — 0 to 31 bits.

Description: rd ← rt >> sa (logical)

The doubleword contents of GPR *rt* are shifted right, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 0 to 31 is specified by *sa*.

Restrictions:

None

Operation: 64-bit processors

s ← 0 || sa
GPR[rd]← 0^s || GPR[rt]$_{63..s}$

Exceptions:

Reserved Instruction

DSRL32 Doubleword Shift Right Logical Plus 32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSRL32 1 1 1 1 1 0
6	5	5	5	5	6

Format: DSRL32 rd, rt, sa **MIPS III**

Purpose: To logical right shift a doubleword by a fixed amount — 32 to 63 bits.

Description: rd ← rt >> (sa+32) (logical)

The 64-bit doubleword contents of GPR *rt* are shifted right, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 32 to 63 is specified by *sa*+32.

Restrictions:

None

Operation: 64- bit processors

```
s          ← 1 || sa          /* 32+sa */
GPR[rd]← 0^s || GPR[rt]_63..s
```

$s \leftarrow 1 \,||\, sa$ /* 32+sa */
$GPR[rd] \leftarrow 0^{s} \,||\, GPR[rt]_{63..s}$

Exceptions:

Reserved Instruction

Doubleword Shift Right Logical Variable DSRLV

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	DSRLV 010110
6	5	5	5	5	6

Format: DSRLV rd, rt, rs **MIPS III**

Purpose: To logical right shift a doubleword by a variable number of bits.

Description: rd ← rt >> rs (logical)

The 64-bit doubleword contents of GPR *rt* are shifted right, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 0 to 63 is specified by the low-order six bits in GPR *rs.*

Restrictions:

None

Operation: 64- bit processors

$s \leftarrow GPR[rs]_{5..0}$
$GPR[rd] \leftarrow 0^{s} \mid\mid GPR[rt]_{63..s}$

Exceptions:

Reserved Instruction

DSUB

Doubleword Subtract

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DSUB 1 0 1 1 1 0
6	5	5	5	5	6

Format: DSUB rd, rs, rt **MIPS III**

Purpose: To subtract 64-bit integers; trap if overflow.

Description: rd ← rs - rt

The 64-bit doubleword value in GPR *rt* is subtracted from the 64-bit value in GPR *rs* to produce a 64-bit result. If the subtraction results in 64-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into GPR *rd*.

Restrictions:

None

Operation: 64- bit processors

```
temp ← GPR[rs] – GPR[rt]
if (64_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp
endif
```

Exceptions:

Integer Overflow
Reserved Instruction

Programming Notes:

DSUBU performs the same arithmetic operation but, does not trap on overflow.

DSUBU

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DSUBU 1 0 1 1 1 1
6	5	5	5	5	6

Format: DSUBU rd, rs, rt **MIPS III**

Purpose: To subtract 64-bit integers.

Description: rd ← rs - rt

The 64-bit doubleword value in GPR *rt* is subtracted from the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation: 64- bit processors

GPR[rd] ← GPR[rs] – GPR[rt]

Exceptions:

Reserved Instruction

Programming Notes:

The term "unsigned" in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as "C" language arithmetic.

J — Jump

31–26	25–0
J 000010	instr_index
6	26

Format: J target — MIPS I

Purpose: To branch within the current 256 MB aligned region.

Description:

This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction following the jump, in the branch delay slot, before jumping.

Restrictions:

None

Operation:

I:
I+1: $PC \leftarrow PC_{GPRLEN..28} \mid\mid instr\_index \mid\mid 0^2$

Exceptions:

None

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch to anywhere in the region from anywhere in the region which a signed relative offset would not allow.

This definition creates the boundary case where the branch instruction is in the last word of a 256 MB region and can therefore only branch to the following 256 MB region containing the branch delay slot.

31 26	25 0
JAL 0 0 0 0 1 1	instr_index
6	26

Format: JAL target **MIPS I**

Purpose: To procedure call within the current 256 MB aligned region.

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction following the jump, in the branch delay slot, before jumping.

Restrictions:

None

Operation:

I: $GPR[31] \leftarrow PC + 8$
I+ 1 : $PC \leftarrow PC_{GPRLEN..28} \,||\, instr\_index \,||\, 0^2$

Exceptions:

None

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch to anywhere in the region from anywhere in the region which a signed relative offset would not allow.

This definition creates the boundary case where the branch instruction is in the last word of a 256 MB region and can therefore only branch to the following 256 MB region containing the branch delay slot.

JALR

Jump And Link Register

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	0 0 0 0 0	rd	0 0 0 0 0	JALR 0 0 1 0 0 1
6	5	5	5	5	6

Format: JALR rs (rd = 31 implied) **MIPS I**

JALR rd, rs

Purpose: To procedure call to an instruction address in a register.

Description: rd ← return_addr, PC ← rs

Place the return address link in GPR *rd*. The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

Jump to the effective target address in GPR *rs*. Execute the instruction following the jump, in the branch delay slot, before jumping.

Restrictions:

Register specifiers *rs* and *rd* must not be equal, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

The effective target address in GPR *rs* must be naturally aligned. If either of the two least-significant bits are not -zero, then an Address Error exception occurs, not for the jump instruction, but when the branch target is subsequently fetched as an instruction.

Operation:

```
I:    temp ← GPR[rs]
      GPR[rd] ← PC + 8
I+1:  PC ← temp
```

Exceptions:

None

Programming Notes:

This is the only branch-and-link instruction that can select a register for the return link; all other link instructions use GPR 31 The default register for *GPR rd*, if omitted in the assembly language instruction, is GPR 31.

Jump Register JR

31–26	25–21	20–6	5–0
SPECIAL 0 0 0 0 0 0	rs	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	JR 0 0 1 0 0 0
6	5	15	6

Format: JR rs — MIPS I

Purpose: To branch to an instruction address in a register.

Description: PC ← rs

Jump to the effective target address in GPR *rs*. Execute the instruction following the jump, in the branch delay slot, before jumping.

Restrictions:

The effective target address in GPR *rs* must be naturally aligned. If either of the two least-significant bits are not -zero, then an Address Error exception occurs, not for the jump instruction, but when the branch target is subsequently fetched as an instruction.

Operation:

```
I:    temp ← GPR[rs]
I+1:  PC ← temp
```

Exceptions:

None

LB

Load Byte

31–26	25–21	20–16	15–0
LB 1 0 0 0 0 0	base	rt	offset
6	5	5	16

Format: LB rt, offset(base) **MIPS I**

Purpose: To load a byte from memory as a signed value.

Description: rt ← memory[base+offset]

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

Operation: 32-bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $\text{pAddr}_{(\text{PSIZE-1})..2}$ || ($\text{pAddr}_{1..0}$ xor ReverseEndian^2)
memword ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← $\text{vAddr}_{1..0}$ xor BigEndianCPU^2
GPR[rt] ← sign_extend($\text{memword}_{7+8*\text{byte}..8*\text{byte}}$)

Operation: 64-bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3}$ || ($\text{pAddr}_{2..0}$ xor ReverseEndian^3)
memdouble ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← $\text{vAddr}_{2..0}$ xor BigEndianCPU^3
GPR[rt] ← sign_extend($\text{memdouble}_{7+8*\text{byte}..8*\text{byte}}$)

Exceptions:

TLB Refill, TLB Invalid
Address Error

Load Byte Unsigned LBU

31–26	25–21	20–16	15–0
LBU 100100	base	rt	offset
6	5	5	16

Format: LBU rt, offset(base) **MIPS I**

Purpose: To load a byte from memory as an unsigned value.

Description: rt ← memory[base+offset]

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

Operation: 32- bit processors

$vAddr \leftarrow sign\_extend(offset) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA, LOAD)$
$pAddr \leftarrow pAddr_{PSIZE-1..2} \mid\mid (pAddr_{1..0}\ xor\ ReverseEndian^{2})$
$memword \leftarrow LoadMemory\ (uncached, BYTE, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^{2}$
$GPR[rt] \leftarrow zero\_extend(memword_{7+8^{*}byte..8^{*}byte})$

Operation: 64- bit processors

$vAddr \leftarrow sign\_extend(offset) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA, LOAD)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \mid\mid (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
$memdouble \leftarrow LoadMemory\ (uncached, BYTE, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^{3}$
$GPR[rt] \leftarrow zero\_extend(memdouble_{7+8^{*}byte..8^{*}byte})$

Exceptions:

TLB Refill, TLB Invalid
Address Error

LD **Load Doubleword**

31 — 26	25 — 21	20 — 16	15 — 0
LD 1 1 0 1 1 1	base	rt	offset
6	5	5	16

Format: LD rt, offset(base) **MIPS III**

Purpose: To load a doubleword from memory.

Description: rt ← memory[base+offset]

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If any of the three least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 3 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 64- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{2..0}$) ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
memdouble ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[rt] ← memdouble

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error
Reserved Instruction

Load Doubleword to Coprocessor LDCz

31–26	25–21	20–16	15–0
LDCz 1 1 0 1 z z	base	rt	offset
6	5	5	16

Format: LDC1 rt, offset(base) MIPS II
LDC2 rt, offset(base)

Purpose: To load a doubleword from memory to a coprocessor general register.

Description: rt ← memory[base+offset]

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and made available to coprocessor unit *zz*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications. The usual operation would place the data into coprocessor general register *rt*.

Each MIPS architecture level defines up to 4 coprocessor units, numbered 0 to 3 (see Coprocessor Instructions on page A-11). The opcodes corresponding to coprocessors that are not defined by an architecture level may be used for other instructions.

Restrictions:

Access to the coprocessors is controlled by system software. Each coprocessor has a "coprocessor usable" bit in the System Control coprocessor. The usable bit must be set for a user program to execute a coprocessor instruction. If the usable bit is not set, an attempt to execute the instruction will result in a Coprocessor Unusable exception. An unimplemented coprocessor must never be enabled. The result of executing this instruction for an unimplemented coprocessor when the usable bit is set, is undefined.

This instruction is not available for coprocessor 0, the System Control coprocessor, and the opcode may be used for other instructions.

The effective address must be naturally aligned. If any of the three least-significant bits of the effective address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 3 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32- bit processors

```
vAddr ← sign_extend(offset) + GPR[base]
if (vAddr2..0) ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
memdouble ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
COP_LD (z, rt, memdouble)
```

Operation: 64- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{2..0}$) ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
memdouble ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
COP_LD (z, rt, memdouble)

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error
Reserved Instruction
Coprocessor Unusable

Load Doubleword Left **LDL**

31–26	25–21	20–16	15–0
LDL 0 1 1 0 1 0	base	rt	offset
6	5	5	16

Format: LDL rt, offset(base) **MIPS III**

Purpose: To load the most-significant part of a doubleword from an unaligned memory address.

Description: rt ← rt MERGE memory[base+offset]

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of eight consecutive bytes forming a doubleword in memory (*DW*) starting at an arbitrary byte boundary. A part of *DW*, the most-significant one to eight bytes, is in the aligned doubleword containing *EffAddr*. This part of *DW* is loaded appropriately into the most-significant (left) part of GPR *rt* leaving the remainder of GPR *rt* unchanged.

The figure below illustrates this operation for big-endian byte ordering. The eight consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, six bytes, is contained in the aligned doubleword containing the most-significant byte at 2. First, LDL loads these six bytes into the left part of the destination register and leaves the remainder of the destination unchanged. Next, the complementary LDR loads the remainder of the unaligned doubleword.

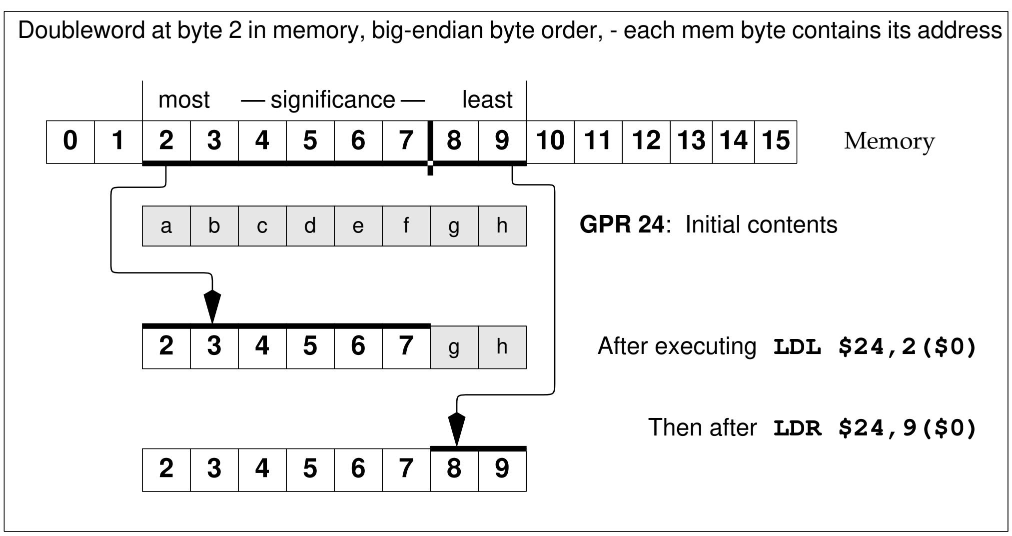


Figure A-2 Unaligned Doubleword Load using LDL and LDR.

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned doubleword, i.e. the low three bits of the address ($vAddr_{2..0}$), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes loaded for every combination of offset and byte ordering.

Table A-28 Bytes Loaded by LDL Instruction

Memory contents and byte offsets ($vAddr_{2..0}$) — most — significance — least

0	1	2	3	4	5	6	7	← big-
I	**J**	**K**	**L**	**M**	**N**	**O**	**P**	
7	6	5	4	3	2	1	0	← little-endian offset

Initial contents of Destination Register — most — significance — least

a	b	c	d	e	f	g	h

Destination register contents after instruction (shaded is unchanged)

Big-endian byte ordering								$vAddr_{2..0}$	Little-endian byte ordering							
I	**J**	**K**	**L**	**M**	**N**	**O**	**P**	0	**P**	b	c	d	e	f	g	h
J	**K**	**L**	**M**	**N**	**O**	**P**	h	1	**O**	**P**	c	d	e	f	g	h
K	**L**	**M**	**N**	**O**	**P**	g	h	2	**N**	**O**	**P**	d	e	f	g	h
L	**M**	**N**	**O**	**P**	f	g	h	3	**M**	**N**	**O**	**P**	e	f	g	h
M	**N**	**O**	**P**	e	f	g	h	4	**L**	**M**	**N**	**O**	**P**	f	g	h
N	**O**	**P**	d	e	f	g	h	5	**K**	**L**	**M**	**N**	**O**	**P**	g	h
O	**P**	c	d	e	f	g	h	6	**J**	**K**	**L**	**M**	**N**	**O**	**P**	h
P	b	c	d	e	f	g	h	7	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**

Restrictions:

None

Operation: 64- bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{(PSIZE-1)..3}$ || ($pAddr_{2..0}$ xor $ReverseEndian^3$)
if BigEndianMem = 0 then
  pAddr ← $pAddr_{(PSIZE-1)..3}$ || 0^3
endif
byte ← $vAddr_{2..0}$ xor $BigEndianCPU^3$
memdouble ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
GPR[rt] ← $memdouble_{7+8*byte..0}$ || $GPR[rt]_{55-8*byte..0}$

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error
Reserved Instruction

Load Doubleword Right **LDR**

31–26	25–21	20–16	15–0
LDR 0 1 1 0 1 1	base	rt	offset
6	5	5	16

Format: LDR rt, offset(base) **MIPS III**

Purpose: To load the least-significant part of a doubleword from an unaligned memory address.

Description: rt ← rt MERGE memory[base+offset]

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of eight consecutive bytes forming a doubleword in memory (*DW*) starting at an arbitrary byte boundary. A part of *DW*, the least-significant one to eight bytes, is in the aligned doubleword containing *EffAddr*. This part of *DW* is loaded appropriately into the least-significant (right) part of GPR *rt* leaving the remainder of GPR *rt* unchanged.

The figure below illustrates this operation for big-endian byte ordering. The eight consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, two bytes, is contained in the aligned doubleword containing the least-significant byte at 9. First, LDR loads these two bytes into the right part of the destination register and leaves the remainder of the destination unchanged. Next, the complementary LDL loads the remainder of the unaligned doubleword.

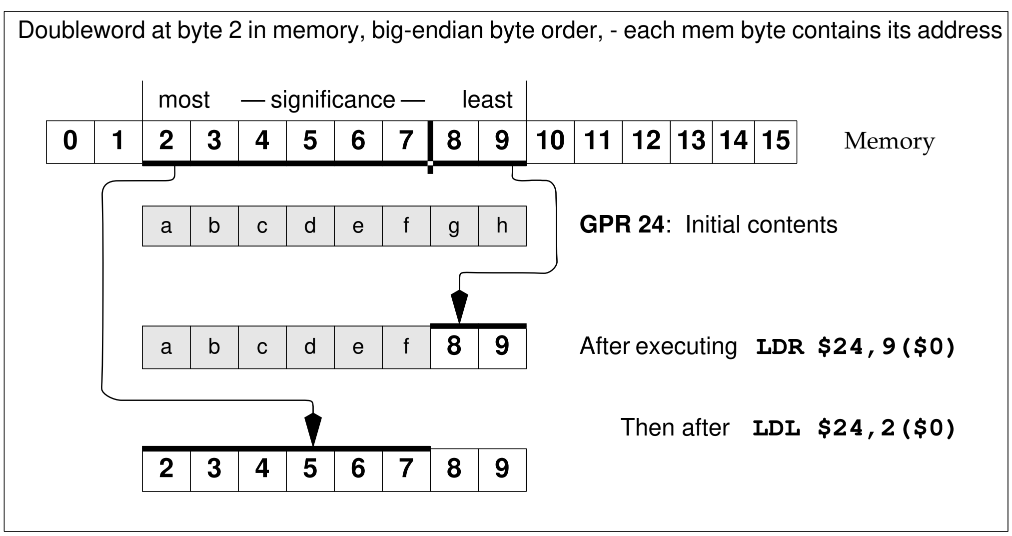


Figure A-3 Unaligned Doubleword Load using LDR and LDL.

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned doubleword, i.e. the low three bits of the address ($vAddr_{2..0}$), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes loaded for every combination of offset and byte ordering.

Table A-29 Bytes Loaded by LDR Instruction

Memory contents and byte offsets ($vAddr_{2..0}$) — most — significance — least

0	1	2	3	4	5	6	7	← big-endian offset
I	**J**	**K**	**L**	**M**	**N**	**O**	**P**	
7	6	5	4	3	2	1	0	← little-endian offset

Initial contents of Destination Register — most — significance — least

a	b	c	d	e	f	g	h

Destination register contents after instruction (shaded is unchanged)

Big-endian byte ordering	$vAddr_{2..0}$	Little-endian byte ordering
a b c d e f g **I**	0	**I J K L M N O P**
a b c d e f **I J**	1	a **I J K L M N O**
a b c d e **I J K**	2	a b **I J K L M N**
a b c d **I J K L**	3	a b c **I J K L M**
a b c **I J K L M**	4	a b c d **I J K L**
a b **I J K L M N**	5	a b c d e **I J K**
a **I J K L M N O**	6	a b c d e f **I J**
I J K L M N O P	7	a b c d e f g **I**

Restrictions:

None

Operation: 64-bit processors

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr(PSIZE-1)..3 || (pAddr2..0 xor ReverseEndian^3)
if BigEndianMem = 1 then
    pAddr ← pAddr(PSIZE-1)..3 || 0^3
endif
byte ← vAddr2..0 xor BigEndianCPU^3
memdouble ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
GPR[rt] ← GPR[rt]63..64-8*byte || memdouble63..8*byte
```

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error
Reserved Instruction

Load Halfword **LH**

31–26	25–21	20–16	15–0
LH 1 0 0 0 0 1	base	rt	offset
6	5	5	16

Format: LH rt, offset(base) **MIPS I**

Purpose: To load a halfword from memory as a signed value.

Description: rt ← memory[base+offset]

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

MIPS IV: The low-order bit of the *offset* field must be zero. If it is not, the result of the instruction is undefined.

Operation: 32- bit processors

$vAddr \leftarrow sign\_extend(offset) + GPR[base]$
if $(vAddr_0) \neq 0$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
$pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, (pAddr_{1..0}$ xor (ReverseEndian || 0))
memword ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
$byte \leftarrow vAddr_{1..0}$ xor (BigEndianCPU || 0)
$GPR[rt] \leftarrow sign\_extend(memword_{15+8*byte..8*byte})$

Operation: 64- bit processors

$vAddr \leftarrow sign\_extend(offset) + GPR[base]$
if $(vAddr_0) \neq 0$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
$pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}$ xor (ReverseEndian || 0))
memdouble ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
$byte \leftarrow vAddr_{2..0}$ xor ($BigEndianCPU^2$ || 0)
$GPR[rt] \leftarrow sign\_extend(memdouble_{15+8*byte..8*byte})$

Exceptions:

TLB Refill , TLB Invalid
Bus Error
Address Error

LHU

Load Halfword Unsigned

31 — 26	25 — 21	20 — 16	15 — 0
LHU 1 0 0 1 0 1	base	rt	offset
6	5	5	16

Format: LHU rt, offset(base) MIPS I

Purpose: To load a halfword from memory as an unsigned value.

Description: rt ← memory[base+offset]

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

MIPS IV: The low-order bit of the *offset* field must be zero. If it is not, the result of the instruction is undefined.

Operation: 32- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_0$) ≠ 0 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor (ReverseEndian || 0))
memword ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
byte ← $vAddr_{1..0}$ xor (BigEndianCPU || 0)
GPR[rt] ← zero_extend($memword_{15+8*byte..8*byte}$)

Operation: 64- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_0$) ≠ 0 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor ($ReverseEndian^2$ || 0))
memdouble ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
byte ← $vAddr_{2..0}$ xor ($BigEndianCPU^2$ || 0)
GPR[rt] ← zero_extend($memdouble_{15+8*byte..8*byte}$)

Exceptions:

TLB Refill, TLB Invalid
Address Error

Load Linked Word **LL**

31–26	25–21	20–16	15–0
LL 1 1 0 0 0 0	base	rt	offset
6	5	5	16

Format: LL rt, offset(base) **MIPS II**

Purpose: To load a word from memory for an atomic read-modify-write.

Description: rt ← memory[base+offset]

The LL and SC instructions provide primitives to implement atomic Read-Modify-Write (RMW) operations for cached memory locations.

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and written into GPR *rt*. This begins a RMW sequence on the current processor.

There is one active RMW sequence per processor. When an LL is executed it starts the active RMW sequence replacing any other sequence that was active.

The RMW sequence is completed by a subsequent SC instruction that either completes the RMW sequence atomically and succeeds, or does not and fails. See the description of SC for a list of events and conditions that cause the SC to fail and an example instruction sequence using LL and SC.

Executing LL on one processor does not cause an action that, by itself, would cause an SC for the same block to fail on another processor.

An execution of LL does not have to be followed by execution of SC; a program is free to abandon the RMW sequence without attempting a write.

Restrictions:

The addressed location must be cached; if it is not, the result is undefined (see Memory Access Types on page A-12).

The effective address must be naturally aligned. If either of the two least-significant bits of the effective address are non-zero an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32-bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) ≠ 0^2 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
LLbit ← 1

Operation: 64-bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) ≠ 0^2 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || 0^2))
memdouble ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
byte ← $vAddr_{2..0}$ xor (BigEndianCPU || 0^2)
GPR[rt] ← sign_extend($memdouble_{31+8*byte..8*byte}$)
LLbit ← 1

Exceptions:

TLB Refill, TLB Invalid
Address Error
Reserved Instruction

Programming Notes:

There is no Load Linked Word Unsigned operation corresponding to Load Word Unsigned.

Implementation Notes:

An LL on one processor must not take action that, by itself, would cause an SC for the same block on another processor to fail. If an implementation depends on retaining the data in cache during the RMW sequence, cache misses caused by LL must not fetch data in the exclusive state, thus removing it from the cache, if it is present in another cache.

31–26	25–21	20–16	15–0
LLD 1 1 0 1 0 0	base	rt	offset
6	5	5	16

Format: LLD rt, offset(base) **MIPS III**

Purpose: To load a doubleword from memory for an atomic read-modify-write.

Description: rt ← memory[base+offset]

The LLD and SCD instructions provide primitives to implement atomic Read-Modify-Write (RMW) operations for cached memory locations.

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and written into GPR *rt*. This begins a RMW sequence on the current processor.

There is one active RMW sequence per processor. When an LLD is executed it starts the active RMW sequence replacing any other sequence that was active.

The RMW sequence is completed by a subsequent SCD instruction that either completes the RMW sequence atomically and succeeds, or does not and fails. See the description of SCD for a list of events and conditions that cause the SCD to fail and an example instruction sequence using LLD and SCD.

Executing LLD on one processor does not cause an action that, by itself, would cause an SCD for the same block to fail on another processor.

An execution of LLD does not have to be followed by execution of SCD; a program is free to abandon the RMW sequence without attempting a write.

Restrictions:

The addressed location must be cached; if it is not, the result is undefined (see Memory Access Types on page A-12).

The effective address must be naturally aligned. If either of the three least-significant bits of the effective address are non-zero an Address Error exception occurs.

MIPS IV: The low-order 3 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

LLD **Load Linked Doubleword**

Operation: 64-bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{2..0}$) ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
memdouble ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[rt] ← memdouble
LLbit ← 1

Exceptions:

TLB Refill, TLB Invalid
Address Error
Reserved Instruction

Programming Notes:

Implementation Notes:

An LLD on one processor must not take action that, by itself, would cause an SCD for the same block on another processor to fail. If an implementation depends on retaining the data in cache during the RMW sequence, cache misses caused by LLD must not fetch data in the exclusive state, thus removing it from the cache, if it is present in another cache.

Load Upper Immediate **LUI**

31 26	25 21	20 16	15 0
LUI 0 0 1 1 1 1	0 0 0 0 0 0	rt	immediate
6	5	5	16

Format: LUI rt, immediate **MIPS I**

Purpose: To load a constant into the upper half of a word.

Description: rt ← immediate || 0^{16}

The 16-bit *immediate* is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is sign-extended and placed into GPR *rt*.

Restrictions:

None

Operation:

GPR[rt] ← sign_extend(immediate || 0^{16})

Exceptions:

None

LW

Load Word

31–26	25–21	20–16	15–0
LW 100011	base	rt	offset
6	5	5	16

Format: LW rt, offset(base) **MIPS I**

Purpose: To load a word from memory as a signed value.

Description: rt ← memory[base+offset]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) ≠ 0^2 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword

Operation: 64- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) ≠ 0^2 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || 0^2))
memdouble ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
byte ← $vAddr_{2..0}$ xor (BigEndianCPU || 0^2)
GPR[rt] ← sign_extend($memdouble_{31+8*byte..8*byte}$)

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error

31–26	25–21	20–16	15–0
LWCz 1 1 0 0 z z	base	rt	offset
6	5	5	16

Format: LWC1 rt, offset(base) MIPS I
LWC2 rt, offset(base)
LWC3 rt, offset(base)

Purpose: To load a word from memory to a coprocessor general register.

Description: rt ← memory[base+offset]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and made available to coprocessor unit *zz*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The manner in which each coprocessor uses the data is defined by the individual coprocessor specification. The usual operation would place the data into coprocessor general register *rt*.

Each MIPS architecture level defines up to 4 coprocessor units, numbered 0 to 3 (see Coprocessor Instructions on page A-11). The opcodes corresponding to coprocessors that are not defined by an architecture level may be used for other instructions.

Restrictions:

Access to the coprocessors is controlled by system software. Each coprocessor has a "coprocessor usable" bit in the System Control coprocessor. The usable bit must be set for a user program to execute a coprocessor instruction. If the usable bit is not set, an attempt to execute the instruction will result in a Coprocessor Unusable exception. An unimplemented coprocessor must never be enabled. The result of executing this instruction for an unimplemented coprocessor when the usable bit is set, is undefined.

This instruction is not available for coprocessor 0, the System Control coprocessor, and the opcode may be used for other instructions.

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32- bit processors

```
I:   vAddr ← sign_extend(offset) + GPR[base]
     if (vAddr1..0) ≠ 0^2 then SignalException(AddressError) endif
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
      memword ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
I+1: COP_LW (z, rt, memword)
```

Operation: 64- bit processors

vAddr ← sign_extend(offset) + GPR[base}
if ($vAddr_{1..0}$) ≠ 0^2 then SignalException(AddressError) endif
(pAddr, uncached)← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || 0^2))
memdouble ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
byte ← $vAddr_{2..0}$ xor (BigEndianCPU || 0^2)
memword ← $memdouble_{31+8*byte..8*byte}$
COP_LW (z, rt, memdouble)

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error
Coprocessor Unusable

Load Word Left LWL

31–26	25–21	20–16	15–0
LWL 100010	base	rt	offset
6	5	5	16

Format: LWL rt, offset(base) **MIPS I**

Purpose: To load the most-significant part of a word as a signed value from an unaligned memory address.

Description: rt ← rt MERGE memory[base+offset]

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of four consecutive bytes forming a word in memory (*W*) starting at an arbitrary byte boundary. A part of *W*, the most-significant one to four bytes, is in the aligned word containing *EffAddr*. This part of *W* is loaded into the most-significant (left) part of the word in GPR *rt*. The remaining least-significant part of the word in GPR *rt* is unchanged.

If GPR *rt* is a 64-bit register, the destination word is the low-order word of the register. The loaded value is treated as a signed value; the word sign bit (bit 31) is always loaded from memory and the new sign bit value is copied into bits 63..32.

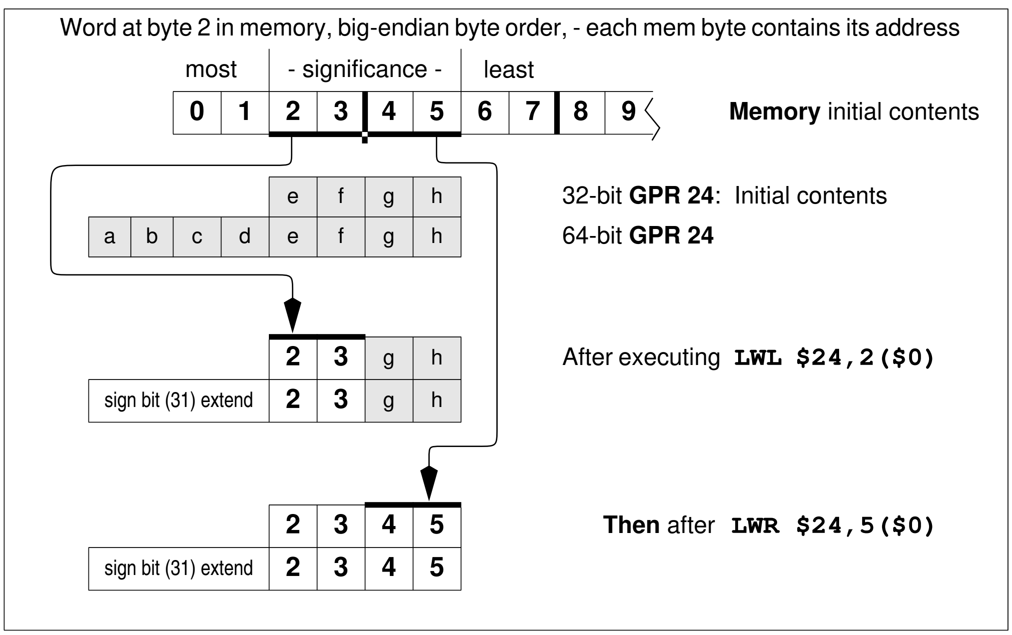


Figure A-4 Unaligned Word Load using LWL and LWR.

The figure above illustrates this operation for big-endian byte ordering for 32-bit and 64-bit registers. The four consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, two bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these two bytes into the left part of the destination register word and leaves the right part of the destination word unchanged. Next, the complementary LWR loads the remainder of the unaligned word.

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, i.e. the low two bits of the address ($vAddr_{1..0}$), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes loaded for every combination of offset and byte ordering.

Table A-30 Bytes Loaded by LWL Instruction

Memory contents and byte offsets

0	1	2	3	← big-endian
I	J	K	L	offset ($vAddr_{1..0}$)
3	2	1	0	← little-endian
most			least	

— significance —

Initial contents of Dest Register

64-bit register

a	b	c	d	e	f	g	h

most — significance — least

32-bit register

e	f	g	h

Destination 64-bit register contents after instruction (shaded is unchanged)

Big-endian byte ordering					$vAddr_{1..0}$	Little-endian byte ordering				
sign bit (31) extended	**I**	**J**	**K**	**L**	0	sign bit (31) extended	**L**	f	g	h
sign bit (31) extended	**J**	**K**	**L**	h	1	sign bit (31) extended	**K**	**L**	g	h
sign bit (31) extended	**K**	**L**	g	h	2	sign bit (31) extended	**J**	**K**	**L**	h
sign bit (31) extended	**L**	f	g	h	3	sign bit (31) extended	**I**	**J**	**K**	**L**

The word sign (31) is always loaded and the value is copied into bits 63..32.

32-bit register

Big-endian				$vAddr_{1..0}$	Little-endian			
I	**J**	**K**	**L**	0	**L**	f	g	h
J	**K**	**L**	h	1	**K**	**L**	g	h
K	**L**	g	h	2	**J**	**K**	**L**	h
L	f	g	h	3	**I**	**J**	**K**	**L**

The unaligned loads, LWL and LWR, are exceptions to the load-delay scheduling restriction in the MIPS I architecture. An unaligned load instruction to GPR *rt* that immediately follows another load to GPR *rt* can “read” the loaded data. It will correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction.

Restrictions:

MIPS I scheduling restriction: The loaded data is not available for use by the following instruction. The instruction immediately following this one, unless it is an unaligned load (LWL, LWR), may not use GPR *rt* as a source register. If this restriction is violated, the result of the operation is undefined.

Operation: 32- bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $\text{pAddr}_{(\text{PSIZE-1})..2}$ || ($\text{pAddr}_{1..0}$ xor ReverseEndian^2)
if BigEndianMem = 0 then
    pAddr ← $\text{pAddr}_{(\text{PSIZE-1})..2}$ || 0^2
endif
byte ← $\text{vAddr}_{1..0}$ xor BigEndianCPU^2
memword ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
GPR[rt] ← $\text{memword}_{7+8*\text{byte}..0}$ || $\text{GPR[rt]}_{23-8*\text{byte}..0}$

Operation: 64- bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $\text{pAddr}_{(\text{PSIZE-1})..3}$ || ($\text{pAddr}_{2..0}$ xor ReverseEndian^3)
if BigEndianMem = 0 then
    pAddr ← $\text{pAddr}_{(\text{PSIZE-1})..3}$ || 0^3
endif
byte ← 0 || ($\text{vAddr}_{1..0}$ xor BigEndianCPU^2)
word ← vAddr_2 xor BigEndianCPU
memdouble ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
temp ← $\text{memdouble}_{31+32*\text{word}-8*\text{byte}..32*\text{word}}$ || $\text{GPR[rt]}_{23-8*\text{byte}..0}$
GPR[rt] ← $(\text{temp}_{31})^{32}$ || temp

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error

Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, i.e. zeroing bits 63..32 of the destination register when bit 31 is loaded. See SLL or SLLV for a single-instruction method of propagating the word sign bit in a register into the upper half of a 64-bit register.

LWR

Load Word Right

31–26	25–21	20–16	15–0
LWR 1 0 0 1 1 0	base	rt	offset
6	5	5	16

Format: LWR rt, offset(base) MIPS I

Purpose: To load the least-significant part of a word from an unaligned memory address as a signed value.

Description: rt ← rt MERGE memory[base+offset]

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr)*. *EffAddr* is the address of the least-significant of four consecutive bytes forming a word in memory *(W)* starting at an arbitrary byte boundary. A part of *W*, the least-significant one to four bytes, is in the aligned word containing *EffAddr*. This part of *W* is loaded into the least-significant (right) part of the word in GPR *rt*. The remaining most-significant part of the word in GPR *rt* is unchanged.

If GPR *rt* is a 64-bit register, the destination word is the low-order word of the register. The loaded value is treated as a signed value; if the word sign bit (bit 31) is loaded (i.e. when all four bytes are loaded) then the new sign bit value is copied into bits 63..32. If bit 31 is not loaded then the value of bits 63..32 is implementation dependent; the value is either unchanged or a copy of the current value of bit 31. Executing both LWR and LWL, in either order, delivers in a sign-extended word value in the destination register.

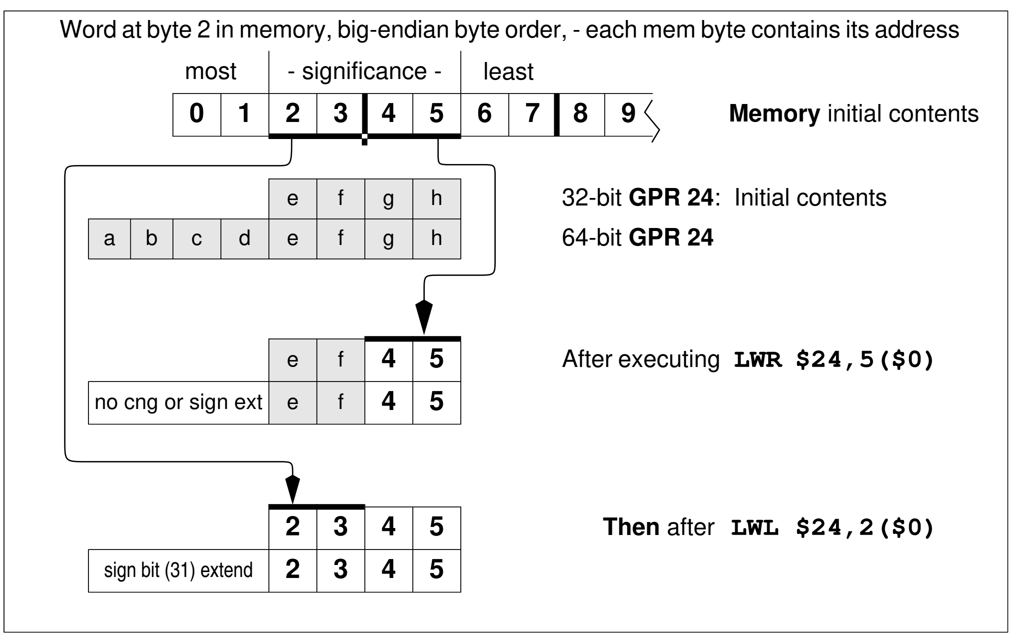


Figure A-5 Unaligned Word Load using LWR and LWL.

The figure above illustrates this operation for big-endian byte ordering for 32-bit and 64-bit registers. The four consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, two bytes, is in the aligned word containing the least-significant byte at 5. First, LWR loads these two bytes into the right part of the destination register. Next, the complementary LWL loads the remainder of the unaligned word.

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, i.e. the low two bits of the address ($vAddr_{1..0}$), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes loaded for every combination of offset and byte ordering.

Table A-31 Bytes Loaded by LWR Instruction

Memory contents and byte offsets

0	1	2	3	← big-endian
I	**J**	**K**	**L**	offset ($vAddr_{1..0}$)
3	2	1	0	← little-endian
most			least	

— significance —

Initial contents of Dest Register

64-bit register

a	b	c	d	e	f	g	h

most — significance — least

32-bit register

e	f	g	h

Destination 64-bit register contents after instruction (shaded is unchanged)

Big-endian byte ordering					$vAddr_{1..0}$	Little-endian byte ordering				
No cng or sign-extend	e	f	g	**I**	0	sign bit (31) extended	**I**	**J**	**K**	**L**
No cng or sign-extend	e	f	**I**	**J**	1	No cng or sign-extend	e	**I**	**J**	**K**
No cng or sign-extend	e	**I**	**J**	**K**	2	No cng or sign-extend	e	f	**I**	**J**
sign bit (31) extended	**I**	**J**	**K**	**L**	3	No cng or sign-extend	e	f	g	**I**

When the word sign bit (31) is loaded, its value is copied into bits 63..32. When it is not loaded, the behavior is implementation specific. Bits 63..32 are either unchanged or a the value of the unloaded bit 31 is copied into them.

32-bit register

big-endian				$vAddr_{1..0}$	little-endian			
e	f	g	**I**	0	**I**	**J**	**K**	**L**
e	f	**I**	**J**	1	e	**I**	**J**	**K**
e	**I**	**J**	**K**	2	e	f	**I**	**J**
I	**J**	**K**	**L**	3	e	f	g	**I**

The unaligned loads, LWL and LWR, are exceptions to the load-delay scheduling restriction in the MIPS I architecture. An unaligned load to GPR *rt* that immediately follows another load to GPR *rt* can “read” the loaded data. It will correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction.

Restrictions:

MIPS I scheduling restriction: The loaded data is not available for use by the following instruction. The instruction immediately following this one, unless it is an unaligned load (LWL, LWR), may not use GPR *rt* as a source register. If this restriction is violated, the result of the operation is undefined.

Restrictions:

None

Operation: 32-bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{(PSIZE-1)..2}$ || ($pAddr_{1..0}$ xor $ReverseEndian^2$)
if BigEndianMem = 0 then
    pAddr ← $pAddr_{(PSIZE-1)..2}$ || 0^2
endif
byte ← $vAddr_{1..0}$ xor $BigEndianCPU^2$
memword ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
GPR[rt] ← $memword_{31..32-8*byte}$ || $GPR[rt]_{31-8*byte..0}$

Operation: 64-bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{(PSIZE-1)..3}$ || ($pAddr_{2..0}$ xor $ReverseEndian^3$)
if BigEndianMem = 1 then
    pAddr ← $pAddr_{(PSIZE-1)..3}$ || 0^3
endif
byte ← $vAddr_{1..0}$ xor $BigEndianCPU^2$
word ← $vAddr_2$ xor BigEndianCPU
memdouble ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
temp ← $GPR[rt]_{31..32-8*byte}$ || $memdouble_{31+32*word..32*word+8*byte}$
if byte = 4 then
    utemp ← $(temp_{31})^{32}$ /* loaded bit 31, must sign extend */
else
    one of the following two behaviors:
        utemp ← $GPR[rt]_{63..32}$ /* leave what was there alone */
        utemp ← $(GPR[rt]_{31})^{32}$ /* sign-extend bit 31 */
endif
GPR[rt] ← utemp || temp

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error

Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, i.e. zeroing bits 63..32 of the destination register when bit 31 is loaded. See SLL or SLLV for a single-instruction method of propagating the word sign bit in a register into the upper half of a 64-bit register.

LWU

Load Word Unsigned

31–26	25–21	20–16	15–0
LWU 1 0 0 1 1 1	base	rt	offset
6	5	5	16

Format: LWU rt, offset(base) **MIPS III**

Purpose: To load a word from memory as an unsigned value.

Description: rt ← memory[base+offset]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 64- bit processors

$\text{vAddr} \leftarrow \text{sign\_extend(offset) + GPR[base]}$
$\text{if (vAddr}_{1..0}) \neq 0^2 \text{ then SignalException(AddressError) endif}$
$\text{(pAddr, uncached)} \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)}$
$\text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE-1..3}} \text{ || (pAddr}_{2..0} \text{ xor (ReverseEndian || } 0^2\text{))}$
$\text{memdouble} \leftarrow \text{LoadMemory (uncached, WORD, pAddr, vAddr, DATA)}$
$\text{byte} \leftarrow \text{vAddr}_{2..0} \text{ xor (BigEndianCPU || } 0^2\text{)}$
$\text{GPR[rt]} \leftarrow 0^{32} \text{ || memdouble}_{31+8*\text{byte}..8*\text{byte}}$

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error
Reserved Instruction

Move From HI Register MFHI

31–26	25–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0	rd	0 0 0 0 0 0	MFHI 0 1 0 0 0 0
6	10	5	5	6

Format: MFHI rd **MIPS I**

Purpose: To copy the special purpose HI register to a GPR.

Description: rd ← HI

The contents of special register *HI* are loaded into GPR *rd*.

Restrictions:

The two instructions that follow an MFHI instruction must not be instructions that modify the *HI* register: DDIV, DDIVU, DIV, DIVU, DMULT, DMULTU, MTHI, MULT, MULTU. If this restriction is violated, the result of the MFHI is undefined.

Operation:

GPR[rd] ← HI

Exceptions:

None

MFLO

Move From LO Register

31–26	25–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0	rd	0 0 0 0 0 0	MFLO 0 1 0 0 1 0
6	10	5	5	6

Format: MFLO rd **MIPS I**

Purpose: To copy the special purpose LO register to a GPR.

Description: rd ← LO

The contents of special register *LO* are loaded into GPR *rd*.

Restrictions:

The two instructions that follow an MFLO instruction must not be instructions that modify the *LO* register: DDIV, DDIVU, DIV, DIVU, DMULT, DMULTU, MTLO, MULT, MULTU. If this restriction is violated, the result of the MFLO is undefined.

Operation:

GPR[rd] ← LO

Exceptions:

None

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	MOVN 0 0 1 0 1 1
6	5	5	5	5	6

Format: MOVN rd, rs, rt **MIPS IV**

Purpose: To conditionally move a GPR after testing a GPR value.

Description: if (rt ≠ 0) then rd ← rs

If the value in GPR *rt* is not equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

None

Operation:

```
if GPR[rt] ≠ 0 then
    GPR[rd] ← GPR[rs]
endif
```

Exceptions:

Reserved Instruction

Programming Notes:

The nonzero value tested here is the "condition true" result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

MOVZ

Move Conditional on Zero

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	MOVZ 001010
6	5	5	5	5	6

Format: MOVZ rd, rs, rt **MIPS IV**

Purpose: To conditionally move a GPR after testing a GPR value.

Description: if (rt = 0) then rd ← rs

If the value in GPR *rt* is equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

None

Operation:

```
if GPR[rt] = 0 then
    GPR[rd] ← GPR[rs]
endif
```

Exceptions:

Reserved Instruction

Programming Notes:

The zero value tested here is the “condition false” result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

Move To HI Register **MTHI**

31–26	25–21	20–6	5–0
SPECIAL 000000	rs	0 0 0000 0000 0000 00	MTHI 010001
6	5	15	6

Format: MTHI rs **MIPS I**

Purpose: To copy a GPR to the special purpose HI register.

Description: HI ← rs

The contents of GPR *rs* are loaded into special register *HI*.

Restrictions:

If either of the two preceding instructions is MFHI, the result of that MFHI is undefined. Reads of the *HI* or *LO* special registers must be separated from subsequent instructions that write to them by two or more other instructions.

A computed result written to the *HI*/*LO* pair by DDIV, DDIVU, DIV, DIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before another result is written into either *HI* or *LO*. If an MTHI instruction is executed following one of these arithmetic instructions, but before a MFLO or MFHI instruction, the contents of *LO* are undefined. The following example shows this illegal situation:

```
MUL    r2,r4    # start operation that will eventually write to HI,LO
...             # code not containing mfhi or mflo
MTHI   r6
...             # code not containing mflo
MFLO   r3       # this mflo would get an undefined value
```

Operation:

```
I-2:, I-1:HI ← undefined
I:        HI ← GPR[rs]
```

Exceptions:

None

MTLO

Move To LO Register

31 … 26	25 … 21	20 … 6	5 … 0
SPECIAL 0 0 0 0 0 0	rs	0 0 0000 0000 0000 00	MTLO 0 1 0 0 1 1
6	5	15	6

Format: MTLO rs **MIPS I**

Purpose: To copy a GPR to the special purpose LO register.

Description: LO ← rs

The contents of GPR *rs* are loaded into special register *LO.*

Restrictions:

If either of the two preceding instructions is MFLO, the result of that MFLO is undefined. Reads of the *HI* or *LO* special registers must be separated from subsequent instructions that write to them by two or more other instructions.

A computed result written to the *HI*/*LO* pair by DDIV, DDIVU, DIV, DIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before another result is written into either *HI* or *LO*. If an MTLO instruction is executed following one of these arithmetic instructions, but before a MFLO or MFHI instruction, the contents of *HI* are undefined. The following example shows this illegal situation:

```
MUL    r2,r4   # start operation that will eventually write to HI,LO
...            # code not containing mfhi or mflo
MTLO   r6
...            # code not containing mfhi
MFHI   r3      # this mfhi would get an undefined value
```

Operation:

I-2:, I-1: LO ← undefined
I: LO ← GPR[rs]

Exceptions:

None

Multiply Word **MULT**

31–26	25–21	20–16	15–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	MULT 0 1 1 0 0 0
6	5	5	10	6

Format: MULT rs, rt **MIPS I**

Purpose: To multiply 32-bit signed integers.

Description: (LO, HI) ← rs × rt

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register *LO*, and the high-order 32-bit word is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the *HI* or *LO* special registers must be separated from subsequent instructions that write to them by two or more other instructions.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
I-2:, I-1:  LO, HI  ← undefined
I:          prod    ← GPR[rs]31..0 * GPR[rt]31..0
            LO      ← sign_extend(prod31..0)
            H I     ← sign_extend(prod63..32)
```

Exceptions:

None

Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written will wait (interlock) until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

MULTU Multiply Unsigned Word

31–26	25–21	20–16	15–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	MULTU 0 1 1 0 0 1
6	5	5	10	6

Format: MULTU rs, rt **MIPS I**

Purpose: To multiply 32-bit unsigned integers.

Description: (LO, HI) ← rs × rt

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register *LO*, and the high-order 32-bit word is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

If either of the two preceding instructions is MFHI or MFLO, the result of the MFHI or MFLO is undefined. Reads of the *HI* or *LO* special registers must be separated from subsequent instructions that write to them by two or more other instructions.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
I-2:, I-1:   LO, HI  ← undefined
I:           prod    ← (0 || GPR[rs]31..0) * (0 || GPR[rt]31..0)
             LO      ← sign_extend(prod31..0)
             H I     ← sign_extend(prod63..32)
```

Exceptions:

None

Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written will wait (interlock) until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

Not Or NOR

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	NOR 1 0 0 1 1 1
6	5	5	5	5	6

Format: NOR rd, rs, rt — MIPS I

Purpose: To do a bitwise logical NOT OR.

Description: rd ← rs NOR rt

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical NOR operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

GPR[rd] ← GPR[rs] nor GPR[rt]

Exceptions:

None

OR

Or

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	OR 1 0 0 1 0 1
6	5	5	5	5	6

Format: OR rd, rs, rt **MIPS I**

Purpose: To do a bitwise logical OR.

Description: rd ← rs OR rt

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical OR operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

GPR[rd] ← GPR[rs] or GPR[rt]

Exceptions:

None

Or Immediate ORI

31 … 26	25 … 21	20 … 16	15 … 0
ORI 0 0 1 1 0 1	rs	rt	immediate
6	5	5	16

Format: ORI rt, rs, immediate **MIPS I**

Purpose: To do a bitwise logical OR with a constant.

Description: rd ← rs OR immediate

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical OR operation. The result is placed into GPR *rt*.

Restrictions:

None

Operation:

GPR[rt] ← zero_extend(immediate) or GPR[rs]

Exceptions:

None

PREF

Prefetch

31–26	25–21	20–16	15–0
PREF 110011	base	hint	offset
6	5	5	16

Format: PREF hint, offset(base) MIPS IV

Purpose: To prefetch data from memory.

Description: prefetch_memory(base+offset)

PREF adds the 16-bit signed *offset* to the contents of GPR *base* to form an effective byte address. It advises that data at the effective address may be used in the near future. The *hint* field supplies information about the way that the data is expected to be used.

PREF is an advisory instruction. It may change the performance of the program. For all *hint* values and all effective addresses, it neither changes architecturally-visible state nor alters the meaning of the program. An implementation may do nothing when executing a PREF instruction.

If MIPS IV instructions are supported and enabled, PREF does not cause addressing-related exceptions. If it raises an exception condition, the exception condition is ignored. If an addressing-related exception condition is raised and ignored, no data will be prefetched, Even if no data is prefetched in such a case, some action that is not architecturally-visible, such as writeback of a dirty cache line, might take place.

PREF will never generate a memory operation for a location with an uncached memory access type (see Memory Access Types on page A-12).

If PREF results in a memory operation, the memory access type used for the operation is determined by the memory access type of the effective address, just as it would be if the memory operation had been caused by a load or store to the effective address.

PREF enables the processor to take some action, typically prefetching the data into cache, to improve program performance. The action taken for a specific PREF instruction is both system and context dependent. Any action, including doing nothing, is permitted that does not change architecturally-visible state or alter the meaning of a program. It is expected that implementations will either do nothing or take an action that will increase the performance of the program.

For a cached location, the expected, and useful, action is for the processor to prefetch a block of data that includes the effective address. The size of the block, and the level of the memory hierarchy it is fetched into are implementation specific.

The *hint* field supplies information about the way the data is expected to be used. No *hint* value causes an action that modifies architecturally-visible state. A processor may use a *hint* value to improve the effectiveness of the prefetch action. The defined *hint* values and the recommended prefetch action are shown in the table below. The *hint* table may be extended in future implementations.

Table A-32 Values of Hint Field for Prefetch Instruction

Value	Name	Data use and desired prefetch action
0	load	Data is expected to be loaded (not modified). Fetch data as if for a load.
1	store	Data is expected to be stored or modified. Fetch data as if for a store.
2-3		Not yet defined.
4	load_streamed	Data is expected to be loaded (not modified) but not reused extensively; it will “stream” through cache. Fetch data as if for a load and place it in the cache so that it will not displace data prefetched as “retained”.
5	store_streamed	Data is expected to be stored or modified but not reused extensively; it will “stream” through cache. Fetch data as if for a store and place it in the cache so that it will not displace data prefetched as “retained”.
6	load_retained	Data is expected to be loaded (not modified) and reused extensively; it should be “retained” in the cache. Fetch data as if for a load and place it in the cache so that it will not be displaced by data prefetched as “streamed”.
7	store_retained	Data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Fetch data as if for a store and place it in the cache so that will not be displaced by data prefetched as “streamed”.
8-31		Not yet defined.

Restrictions:

None

Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(uncached, pAddr, vAddr, DATA, hint)
```

Exceptions:

Reserved Instruction

Programming Notes:

Prefetch can not prefetch data from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. It will not cause an exception to prefetch using an address pointer value before the validity of a pointer is determined.

Implementation Notes:

It is recommended that a reserved *hint* field value either cause a default prefetch action that is expected to be useful for most cases of data use, such as the "load" *hint*, or cause the instruction to be treated as a NOP.

Store Byte SB

31–26	25–21	20–16	15–0
SB 1 0 1 0 0 0	base	rt	offset
6	5	5	16

Format: SB rt, offset(base) **MIPS I**

Purpose: To store a byte to memory.

Description: memory[base+offset] ← rt

The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

Operation: 32- bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor $ReverseEndian^2$)
byte ← $vAddr_{1..0}$ xor $BigEndianCPU^2$
dataword ← $GPR[rt]_{31-8*byte..0}$ || 0^{8*byte}
StoreMemory (uncached, BYTE, dataword, pAddr, vAddr, DATA)

Operation: 64- bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor $ReverseEndian^3$)
byte ← $vAddr_{2..0}$ xor $BigEndianCPU^3$
datadouble ← $GPR[rt]_{63-8*byte..0}$ || 0^{8*byte}
StoreMemory (uncached, BYTE, datadouble, pAddr, vAddr, DATA)

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Bus Error
Address Error

31 — 26	25 — 21	20 — 16	15 — 0
SC 1 1 1 0 0 0	base	rt	offset
6	5	5	16

Format: SC rt, offset(base) MIPS II

Purpose: To store a word to memory to complete an atomic read-modify-write.

Description: if (atomic_update) then memory[base+offset] ← rt, rt ← 1 else rt ← 0

The LL and SC instructions provide primitives to implement atomic Read-Modify-Write (RMW) operations for cached memory locations.

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The SC completes the RMW sequence begun by the preceding LL instruction executed on the processor. If it would complete the RMW sequence atomically, then the least-significant 32-bit word of GPR *rt* is stored into memory at the location specified by the aligned effective address and a one, indicating success, is written into GPR *rt*. Otherwise, memory is not modified and a zero, indicating failure, is written into GPR *rt*.

If any of the following events occurs between the execution of LL and SC, the SC will fail:

- A coherent store is completed by another processor or coherent I/O module into the block of physical memory containing the word. The size and alignment of the block is implementation dependent. It is at least one word and is at most the minimum page size.
- An exception occurs on the processor executing the LL/SC.
  An implementation may detect “an exception” in one of three ways:
  1) Detect exceptions and fail when an exception occurs.
  2) Fail after the return-from-interrupt instruction (RFE or ERET) is executed.
  3) Do both 1 and 2.

If any of the following events occurs between the execution of LL and SC, the SC may succeed or it may fail; the success or failure is unpredictable. Portable programs should not cause one of these events.

- A load, store, or prefetch is executed on the processor executing the LL/SC.
- The instructions executed starting with the LL and ending with the SC do not lie in a 2048-byte contiguous region of virtual memory. The region does not have to be aligned, other than the alignment required for instruction words.

The following conditions must be true or the result of the SC will be undefined:

- Execution of SC must have been preceded by execution of an LL instruction.

- A RMW sequence executed without intervening exceptions must use the same address in the LL and SC. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.

Atomic RMW is provided only for cached memory locations. The extent to which the detection of atomicity operates correctly depends on the system implementation and the memory access type used for the location. See Memory Access Types on page A-12.

MP atomicity: To provide atomic RMW among multiple processors, all accesses to the location must be made with a memory access type of cached coherent.

Uniprocessor atomicity: To provide atomic RMW on a single processor, all accesses to the location must be made with memory access type of either cached noncoherent or cached coherent. All accesses must be to one or the other access type, they may not be mixed.

I/O System: To provide atomic RMW with a coherent I/O system, all accesses to the location must be made with a memory access type of cached coherent. If the I/O system does not use coherent memory operations, then atomic RMW cannot be provided with respect to the I/O reads and writes.

The definition above applies to user-mode operation on all MIPS processors that support the MIPS II architecture. There may be other implementation-specific events, such as privileged CP0 instructions, that will cause an SC instruction to fail in some cases. System programmers using LL/SC should consult implementation-specific documentation.

Restrictions:

The addressed location must have a memory access type of cached noncoherent or cached coherent; if it does not, the result is undefined (see Memory Access Types on page A-12).

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) ≠ 0^2 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
if LLbit then
    StoreMemory (uncached, WORD, dataword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 0^{31} || LLbit

Operation: 64- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if $(vAddr_{1..0}) \neq 0^2$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || 0^2))
byte ← $vAddr_{2..0}$ xor (BigEndianCPU || 0^2)
datadouble ← $GPR[rt]_{63-8*byte..0}$ || 0^{8*byte}
if LLbit then
    StoreMemory (uncached, WORD, datadouble, pAddr, vAddr, DATA)
endif
GPR[rt] ← 0^{63} || LLbit

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error
Reserved Instruction

Programming Notes:

LL and SC are used to atomically update memory locations as shown in the example atomic increment operation below.

```
L1:
    LL      T1, (T0)     # load counter
    ADDI    T2, T1, 1    # increment
    SC      T2, (T0)     # try to store, checking for atomicity
    BEQ     T2, 0, L1    # if not atomic (0), try again
    NOP                  # branch-delay slot
```

Exceptions between the LL and SC cause SC to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, floating-point operations that trap or require software emulation assistance.

LL and SC function on a single processor for cached noncoherent memory so that parallel programs can be run on uniprocessor systems that do not support cached coherent memory access types.

Implementation Notes:

The block of memory that is “locked” for LL/SC is typically the largest cache line in use.

31–26	25–21	20–16	15–0
SCD 1 1 1 1 0 0	base	rt	offset
6	5	5	16

Format: SCD rt, offset(base) **MIPS III**

Purpose: To store a doubleword to memory to complete an atomic read-modify-write.

Description: if (atomic_update) then memory[base+offset] ← rt, rt ← 1 else rt ← 0

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The SCD completes the RMW sequence begun by the preceding LLD instruction executed on the processor. If it would complete the RMW sequence atomically, then the 64-bit doubleword of GPR *rt* is stored into memory at the location specified by the aligned effective address and a one, indicating success, is written into GPR *rt*. Otherwise, memory is not modified and a zero, indicating failure, is written into GPR *rt*.

If any of the following events occurs between the execution of LLD and SCD, the SCD will fail:

- A coherent store is completed by another processor or coherent I/O module into the block of physical memory containing the word. The size and alignment of the block is implementation dependent. It is at least one doubleword and is at most the minimum page size.
- An exception occurs on the processor executing the LLD/SCD.
  An implementation may detect “an exception” in one of three ways:
  1) Detect exceptions and fail when an exception occurs.
  2) Fail after the return-from-interrupt instruction (RFE or ERET) is executed.
  3) Do both 1 and 2.

If any of the following events occurs between the execution of LLD and SCD, the SCD may succeed or it may fail; the success or failure is unpredictable. Portable programs should not cause one of these events.

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LLD/SCD.
- The instructions executed starting with the LLD and ending with the SCD do not lie in a 2048-byte contiguous region of virtual memory. The region does not have to be aligned, other than the alignment required for instruction words.

The following conditions must be true or the result of the SCD will be undefined:

- Execution of SCD must have been preceded by execution of an LLD instruction.

- A RMW sequence executed without intervening exceptions must use the same address in the LLD and SCD. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.

Atomic RMW is provided only for memory locations with cached noncoherent or cached coherent memory access types. The extent to which the detection of atomicity operates correctly depends on the system implementation and the memory access type used for the location. See Memory Access Types on page A-12.

MP atomicity : To provide atomic RMW among multiple processors, all accesses to the location must be made with a memory access type of cached coherent.

Uniprocessor atomicity : To provide atomic RMW on a single processor, all accesses to the location must be made with memory access type of either cached noncoherent or cached coherent. All accesses must be to one or the other access type, they may not be mixed.

I/O System : To provide atomic RMW with a coherent I/O system, all accesses to the location must be made with a memory access type of cached coherent. If the I/O system does not use coherent memory operations, then atomic RMW cannot be provided with respect to the I/O reads and writes.

The defemination above applies to user-mode operation on all MIPS processors that support the MIPS III architecture. There may be other implementation-specific events, such as privileged CP0 instructions, that will cause an SCD instruction to fail in some cases. System programmers using LLD/SCD should consult implementation-specific documentation.

Restrictions:

The addressed location must have a memory access type of cached noncoherent or cached coherent; if it does not, the result is undefined (see Memory Access Types on page A-12The 64-bit doubleword of register *rt* is conditionally stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The effective address must be naturally aligned. If any of the three least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 3 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 64- bit processors

```
vAddr ← sign_extend(offset) + GPR[base]
if (vAddr2..0) ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
datadouble ← GPR[rt]
if LLbit then
    StoreMemory (uncached, DOUBLEWORD, datadouble, pAddr, vAddr, DATA)
endif
GPR[rt] ← 0^63 || LLbit
```

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error
Reserved Instruction

Programming Notes:

LLD and SCD are used to atomically update memory locations as shown in the example atomic increment operation below.

```
L1:
    LLD     T1, (T0)     # load counter
    ADDI    T2, T1, 1    # increment
    SCD     T2, (T0)     # try to store, checking for atomicity
    BEQ     T2, 0, L1    # if not atomic (0), try again
    NOP                  # branch-delay slot
```

Exceptions between the LLD and SCD cause SCD to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, floating-point operations that trap or require software emulation assistance.

LLD and SCD function on a single processor for cached noncoherent memory so that parallel programs can be run on uniprocessor systems that do not support cached coherent memory access types.

Implementation Notes:

The block of memory that is “locked” for LLD/SCD is typically the largest cache line in use.

31–26	25–21	20–16	15–0
SD 1 1 1 1 1 1	base	rt	offset
6	5	5	16

Format: SD rt, offset(base) **MIPS III**

Purpose: To store a doubleword to memory.

Description: memory[base+offset] $\leftarrow$ rt

The 64-bit doubleword in GPR *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If any of the three least-significant bits of the effective address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 3 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 64- bit processors

```
vAddr ← sign_extend(offset) + GPR[base]
if (vAddr2..0) ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
datadouble ← GPR[rt]
StoreMemory (uncached, DOUBLEWORD, datadouble, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error
Reserved Instruction

31 — 26	25 — 21	20 — 16	15 — 0
SDCz 1 1 1 1 z z	base	rt	offset
6	5	5	16

Format: SDC1 rt, offset(base) — MIPS II
SDC2 rt, offset(base)

Purpose: To store a doubleword from a coprocessor general register to memory.

Description: memory[base+offset] ← rt

Coprocessor unit *zz* supplies a 64-bit doubleword which is stored at the memory location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The data supplied by each coprocessor is defined by the individual coprocessor specifications. The usual operation would read the data from coprocessor general register *rt*.

Each MIPS architecture level defines up to 4 coprocessor units, numbered 0 to 3 (see Coprocessor Instructions on page A-11). The opcodes corresponding to coprocessors that are not defined by an architecture level may be used for other instructions.

Restrictions:

Access to the coprocessors is controlled by system software. Each coprocessor has a "coprocessor usable" bit in the System Control coprocessor. The usable bit must be set for a user program to execute a coprocessor instruction. If the usable bit is not set, an attempt to execute the instruction will result in a Coprocessor Unusable exception. An unimplemented coprocessor must never be enabled. The result of executing this instruction for an unimplemented coprocessor when the usable bit is set, is undefined.

This instruction is not defined for coprocessor 0, the System Control coprocessor, and the opcode may be used for other instructions.

The effective address must be naturally aligned. If any of the three least-significant bits of the effective address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 3 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{2..0}$) ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
datadouble ← COP_SD(z, rt)
StoreMemory (uncached, DOUBLEWORD, datadouble, pAddr, vAddr, DATA)

Operation: 64- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{2..0}$) ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
datadouble ← COP_SD(z, rt)
StoreMemory (uncached, DOUBLEWORD, datadouble, pAddr, vAddr, DATA)

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error
Reserved Instruction
Coprocessor Unusable

Store Doubleword Left **SDL**

31–26	25–21	20–16	15–0
SDL 101100	base	rt	offset
6	5	5	16

Format: SDL rt, offset(base) **MIPS III**

Purpose: To store the most-significant part of a doubleword to an unaligned memory address.

Description: memory[base+offset] ← Some_Bytes_From rt

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of eight consecutive bytes forming a doubleword in memory (*DW*) starting at an arbitrary byte boundary. A part of *DW*, the most-significant one to eight bytes, is in the aligned doubleword containing *EffAddr*. The same number of most-significant (left) bytes of GPR *rt* are stored into these bytes of *DW*.

The figure below illustrates this operation for big-endian byte ordering. The eight consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, six bytes, is contained in the aligned doubleword containing the most-significant byte at 2. First, SDL stores the six most-significant bytes of the source register into these bytes in memory. Next, the complementary SDR instruction stores the remainder of *DW*.

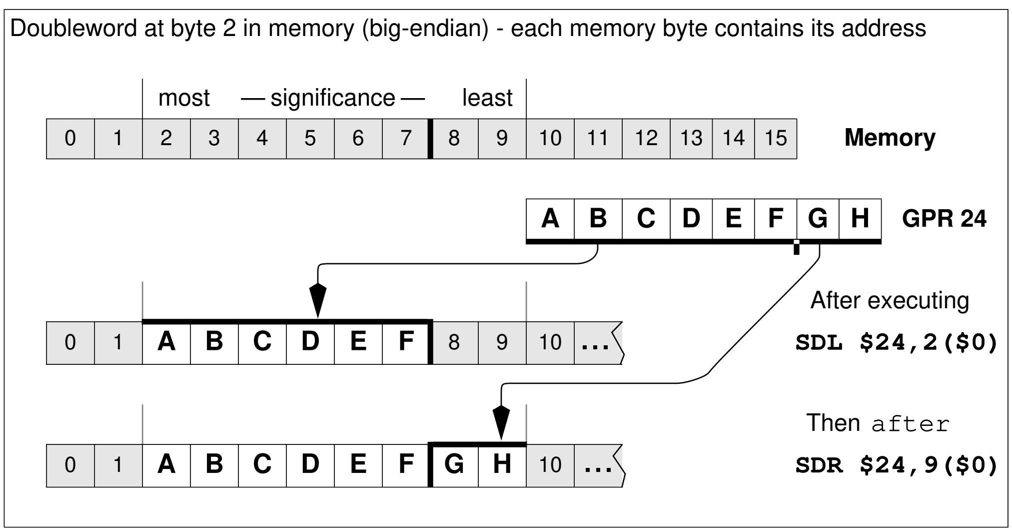


Figure A-6 Unaligned Doubleword Store with SDL and SDR

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned doubleword, i.e. the low three bits of the address ($vAddr_{2..0}$), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes stored for every combination of offset and byte ordering.

Table A-33 Bytes Stored by SDL Instruction

Initial Memory contents and byte offsets (most — significance — least)

0	1	2	3	4	5	6	7	← big-
i	j	k	l	m	n	o	p	
7	6	5	4	3	2	1	0	← little-endian

Contents of Source Register (most — significance — least)

A	B	C	D	E	F	G	H

Memory contents after instruction (shaded is unchanged)

Big-endian byte ordering								$vAddr_{2..0}$	Little-endian byte ordering							
A	B	C	D	E	F	G	H	0	i	j	k	l	m	n	o	A
i	A	B	C	D	E	F	G	1	i	j	k	l	m	n	A	B
i	j	A	B	C	D	E	F	2	i	j	k	l	m	A	B	C
i	j	k	A	B	C	D	E	3	i	j	k	l	A	B	C	D
i	j	k	l	A	B	C	D	4	i	j	k	A	B	C	D	E
i	j	k	l	m	A	B	C	5	i	j	A	B	C	D	E	F
i	j	k	l	m	n	A	B	6	i	A	B	C	D	E	F	G
i	j	k	l	m	n	o	A	7	A	B	C	D	E	F	G	H

Restrictions:

None

Operation: 64- bit processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{(PSIZE-1)..3}$ || ($pAddr_{2..0}$ xor $ReverseEndian^3$)
If BigEndianMem = 0 then
    pAddr ← $pAddr_{(PSIZE-1)..3}$ || 0^3
endif
byte ← $vAddr_{2..0}$ xor $BigEndianCPU^3$
datadouble ← $0^{56-8*byte}$ || $GPR[rt]_{63..56-8*byte}$
StoreMemory (uncached, byte, datadouble, pAddr, vAddr, DATA)

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Bus Error
Address Error
Reserved Instruction

Store Doubleword Right SDR

31 26	25 21	20 16	15 0
SDR 1 0 1 1 0 1	base	rt	offset
6	5	5	16

Format: SDR rt, offset(base) **MIPS III**

Purpose: To store the least-significant part of a doubleword to an unaligned memory address.

Description: memory[base+offset] ← Some_Bytes_From rt

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of eight consecutive bytes forming a doubleword in memory (*DW*) starting at an arbitrary byte boundary. A part of *DW*, the least-significant one to eight bytes, is in the aligned doubleword containing *EffAddr*. The same number of least-significant (right) bytes of GPR *rt* are stored into these bytes of *DW*.

The figure below illustrates this operation for big-endian byte ordering. The eight consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, two bytes, is contained in the aligned doubleword containing the least-significant byte at 9. First, SDR stores the two least-significant bytes of the source register into these bytes in memory. Next, the complementary SDL stores the remainder of *DW*.

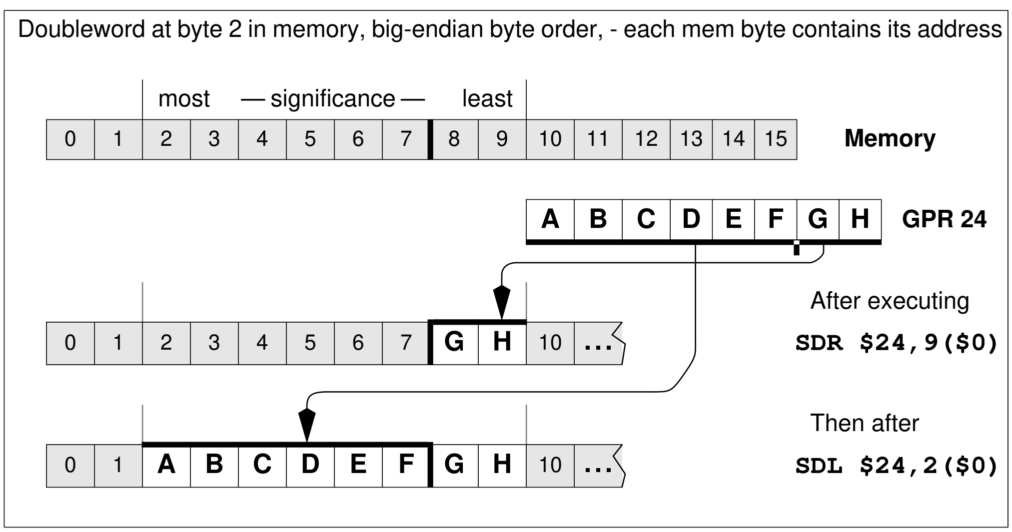


Figure A-7 Unaligned Doubleword Store with SDR and SDL

SDR — Store Doubleword Right

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned doubleword, i.e. the low three bits of the address ($vAddr_{2..0}$), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes stored for every combination of offset and byte ordering.

Table A-34 Bytes Stored by SDR Instruction

Initial Memory contents and byte offsets (most — significance — least)

0	1	2	3	4	5	6	7	← big-
i	j	k	l	m	n	o	p	
7	6	5	4	3	2	1	0	¨ little-endian

Contents of Source Register (most — significance — least)

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**

Memory contents after instruction (shaded is unchanged)

Big-endian byte ordering	$vAddr_{2..0}$	Little-endian byte ordering
H j k l m n o p	0	**A B C D E F G H**
G H k l m n o p	1	**B C D E F G H** p
F G H l m n o p	2	**C D E F G H** o p
E F G H m n o p	3	**D E F G H** n o p
D E F G H n o p	4	**E F G H** m n o p
C D E F G H o p	5	**F G H** l m n o p
B C D E F G H p	6	**G H** k l m n o p
A B C D E F G H	7	**H** j k l m n o p

Restrictions:

None

Operation: 64- bit processors

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr(PSIZE-1)..3 || (pAddr2..0 xor ReverseEndian^3)
If BigEndianMem = 0 then
    pAddr ← pAddr(PSIZE-1)..3 || 0^3
endif
byte ← vAddr1..0 xor BigEndianCPU^3
datadouble ← GPR[rt]63–8*byte || 0^(8*byte)
StoreMemory (uncached, DOUBLEWORD-byte, datadouble, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Bus Error
Address Error
Reserved Instruction

Store Halfword SH

31–26	25–21	20–16	15–0
SH 1 0 1 0 0 1	base	rt	offset
6	5	5	16

Format: SH rt, offset(base) **MIPS I**

Purpose: To store a halfword to memory.

Description: memory[base+offset] ← rt

The least-significant 16-bit halfword of register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

MIPS IV: The low-order bit of the *offset* field must be zero. If it is not, the result of the instruction is undefined.

Operation: 32- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_0$) ≠ 0 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor (ReverseEndian || 0))
byte ← $vAddr_{1..0}$ xor (BigEndianCPU || 0)
dataword ← $GPR[rt]_{31-8*byte..0}$ || 0^{8*byte}
StoreMemory (uncached, HALFWORD, dataword, pAddr, vAddr, DATA)

Operation: 64- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_0$) ≠ 0 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor ($ReverseEndian^2$ || 0))
byte ← $vAddr_{2..0}$ xor ($BigEndianCPU^2$ || 0)
datadouble ← $GPR[rt]_{63-8*byte..0}$ || 0^{8*byte}
StoreMemory (uncached, HALFWORD, datadouble, pAddr, vAddr, DATA)

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error

SLL **Shift Word Left Logical**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	SLL 0 0 0 0 0 0
6	5	5	5	5	6

Format: SLL rd, rt, sa **MIPS I**

Purpose: To left shift a word by a fixed number of bits.

Description: rd ← rt << sa

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeroes into the emptied bits; the word result is placed in GPR *rd*. The bit shift count is specified by *sa.* If *rd* is a 64-bit register, the result word is sign-extended.

Restrictions:

None

Operation:

```
s         ← sa
temp      ← GPR[rt](31-s)..0 || 0^s
GPR[rd]← sign_extend(temp)
```

Exceptions:

None

Programming Notes:

Unlike nearly all other word operations the input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign extends it.

Some assemblers, particularly 32-bit assemblers, treat this instruction with a shift amount of zero as a NOP and either delete it or replace it with an actual NOP.

Shift Word Left Logical Variable SLLV

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	SLLV 000100
6	5	5	5	5	6

Format: SLLV rd, rt, rs **MIPS I**

Purpose: To left shift a word by a variable number of bits.

Description: rd ← rt << rs

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeroes into the emptied bits; the result word is placed in GPR *rd*. The bit shift count is specified by the low-order five bits of GPR *rs*. If *rd* is a 64-bit register, the result word is sign-extended.

Restrictions:

None

Operation:

s ← $GP[rs]_{4..0}$
temp ← $GPR[rt]_{(31-s)..0}$ || 0^s
GPR[rd]← sign_extend(temp)

Exceptions:

None

Programming Notes:

Unlike nearly all other word operations the input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign extends it.

Some assemblers, particularly 32-bit assemblers, treat this instruction with a shift amount of zero as a NOP and either delete it or replace it with an actual NOP.

SLT

Set On Less Than

31 – 26	25 – 21	20 – 16	15 – 11	10 – 6	5 – 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SLT 1 0 1 0 1 0
6	5	5	5	5	6

Format: SLT rd, rs, rt **MIPS I**

Purpose: To record the result of a less-than comparison.

Description: rd ← (rs < rt)

Compare the contents of GPR *rs* and GPR *rt* as signed integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt* the result is 1 (true), otherwise 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```
if GPR[rs] < GPR[rt] then
    GPR[rd] ← 0^(GPRLEN-1) || 1
else
    GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

Set on Less Than Immediate SLTI

31–26	25–21	20–16	15–0
SLTI 0 0 1 0 1 0	rs	rt	immediate
6	5	5	16

Format: SLTI rt, rs, immediate **MIPS I**

Purpose: To record the result of a less-than comparison with a constant.

Description: rt ← (rs < immediate)

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate* the result is 1 (true), otherwise 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```
if GPR[rs] < sign_extend(immediate) then
    GPR[rd] ← 0^(GPRLEN-1) || 1
else
    GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

SLTIU

Set on Less Than Immediate Unsigned

31–26	25–21	20–16	15–0
SLTIU 001011	rs	rt	immediate
6	5	5	16

Format: SLTIU rt, rs, immediate **MIPS I**

Purpose: To record the result of an unsigned less-than comparison with a constant.

Description: rt ← (rs < immediate)

Compare the contents of GPR *rs* and the sign-extended 16-bit *immediate* as unsigned integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate* the result is 1 (true), otherwise 0 (false).

Because the 16-bit *immediate* is sign-extended before comparison, the instruction is able to represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```
    if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
        GPR[rd] ← 0^(GPRLEN-1) || 1
    else
        GPR[rd] ← 0^GPRLEN
    endif
```

Exceptions:

None

Set on Less Than Unsigned **SLTU**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SLTU 1 0 1 0 1 1
6	5	5	5	5	6

Format: SLTU rd, rs, rt **MIPS I**

Purpose: To record the result of an unsigned less-than comparison.

Description: rd ← (rs < rt)

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt* the result is 1 (true), otherwise 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```
    if (0 || GPR[rs]) < (0 || GPR[rt]) then
        GPR[rd] ← 0^(GPRLEN-1) || 1
    else
        GPR[rd] ← 0^GPRLEN
    endif
```

Exceptions:

None

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	SRA 0 0 0 0 1 1
6	5	5	5	5	6

Format: SRA rd, rt, sa **MIPS I**

Purpose: To arithmetic right shift a word by a fixed number of bits.

Description: rd ← rt >> sa (arithmetic)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit shift count is specified by *sa*. If *rd* is a 64-bit register, the result word is sign-extended.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

Operation:

if (NotWordValue(GPR[rt])) then UndefinedResult() endif
s ← sa
temp ← $(GPR[rt]_{31})^{s}$ || $GPR[rt]_{31..s}$
GPR[rd]← sign_extend(temp)

Exceptions:

None

Shift Word Right Arithmetic Variable **SRAV**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SRAV 0 0 0 1 1 1
6	5	5	5	5	6

Format: SRAV rd, rt, rs **MIPS I**

Purpose: To arithmetic right shift a word by a variable number of bits.

Description: rd ← rt >> rs (arithmetic)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit shift count is specified by the low-order five bits of GPR *rs*. If *rd* is a 64-bit register, the result word is sign-extended.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

Operation:

if (NotWordValue(GPR[rt])) then UndefinedResult() endif
s $\leftarrow$ GPR[rs]$_{4..0}$
temp $\leftarrow$ (GPR[rt]$_{31}$)^{s} || GPR[rt]$_{31..s}$
GPR[rd] $\leftarrow$ sign_extend(temp)

Exceptions:

None

SRL

Shift Word Right Logical

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	SRL 0 0 0 0 1 0
6	5	5	5	5	6

Format: SRL rd, rt, sa **MIPS I**

Purpose: To logical right shift a word by a fixed number of bits.

Description: rd ← rt >> sa (logical)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit shift count is specified by *sa.* If *rd* is a 64-bit register, the result word is sign-extended.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

Operation:

```
if (NotWordValue(GPR[rt])) then UndefinedResult() endif
s        ← sa
temp     ← 0^s || GPR[rt]_31..s
GPR[rd]← sign_extend(temp)
```

Exceptions:

None

Shift Word Right Logical Variable SRLV

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SRLV 0 0 0 1 1 0
6	5	5	5	5	6

Format: SRLV rd, rt, rs **MIPS I**

Purpose: To logical right shift a word by a variable number of bits.

Description: rd ← rt >> rs (logical)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit shift count is specified by the low-order five bits of GPR *rs*. If *rd* is a 64-bit register, the result word is sign-extended.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

Operation:

if (NotWordValue(GPR[rt])) then UndefinedResult() endif
s ← $GPR[rs]_{4..0}$
temp ← 0^s || $GPR[rt]_{31..s}$
GPR[rd]← sign_extend(temp)

Exceptions:

None

SUB

Subtract Word

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SUB 1 0 0 0 1 0
6	5	5	5	5	6

Format: SUB rd, rs, rt **MIPS I**

Purpose: To subtract 32-bit integers. If overflow occurs, then trap.

Description: rd ← rs - rt

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* to produce a 32-bit result. If the subtraction results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
temp ← GPR[rs] - GPR[rt]
if (32_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rd] ←temp
endif
```

Exceptions:

Integer Overflow

Programming Notes:

SUBU performs the same arithmetic operation but, does not trap on overflow.

Subtract Unsigned Word SUBU

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SUBU 1 0 0 0 1 1
6	5	5	5	5	6

Format: SUBU rd, rs, rt — MIPS I

Purpose: To subtract 32-bit integers.

Description: rd ← rs - rt

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rd*.

No integer overflow exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
temp ←GPR[rs] - GPR[rt]
GPR[rd] ←temp
```

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as “C” language arithmetic.

SW **Store Word**

31 26	25 21	20 16	15 0
SW 1 0 1 0 1 1	base	rt	offset
6	5	5	16

Format: SW rt, offset(base) **MIPS I**

Purpose: To store a word to memory.

Description: memory[base+offset] $\leftarrow$ rt

The least-significant 32-bit word of register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32- bit Processors

vAddr $\leftarrow$ sign_extend(offset) + GPR[base]
if (vAddr$_{1..0}$) $\neq$ 0^2 then SignalException(AddressError) endif
(pAddr, uncached) $\leftarrow$ AddressTranslation (vAddr, DATA, STORE)
dataword $\leftarrow$ GPR[rt]
StoreMemory (uncached, WORD, dataword, pAddr, vAddr, DATA)

Operation: 64- bit Processors

vAddr $\leftarrow$ sign_extend(offset) + GPR[base]
if (vAddr$_{1..0}$) $\neq$ 0^2 then SignalException(AddressError) endif
(pAddr, uncached) $\leftarrow$ AddressTranslation (vAddr, DATA, STORE)
pAddr $\leftarrow$ pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor (ReverseEndian || 0^2)
byte $\leftarrow$ vAddr$_{2..0}$ xor (BigEndianCPU || 0^2)
datadouble $\leftarrow$ GPR[rt]$_{63-8*byte}$ || 0^{8*byte}
StoreMemory (uncached, WORD, datadouble, pAddr, vAddr, DATA)

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error

Store Word From Coprocessor **SWCz**

31 26	25 21	20 16	15 0
SWCz 1 1 1 0 z z	base	rt	offset
6	5	5	16

Format: SWC1 rt, offset(base) **MIPS I**
SWC2 rt, offset(base)
SWC3 rt, offset(base)

Purpose: To store a word from a coprocessor general register to memory.

Description: memory[base+offset] ← rt

Coprocessor unit *zz* supplies a 32-bit word which is stored at the memory location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The data supplied by each coprocessor is defined by the individual coprocessor specifications. The usual operation would read the data from coprocessor general register *rt*.

Each MIPS architecture level defines up to 4 coprocessor units, numbered 0 to 3 (see Coprocessor Instructions on page A-11). The opcodes corresponding to coprocessors that are not defined by an architecture level may be used for other instructions.

Restrictions:

Access to the coprocessors is controlled by system software. Each coprocessor has a "coprocessor usable" bit in the System Control coprocessor. The usable bit must be set for a user program to execute a coprocessor instruction. If the usable bit is not set, an attempt to execute the instruction will result in a Coprocessor Unusable exception. An unimplemented coprocessor must never be enabled. The result of executing this instruction for an unimplemented coprocessor when the usable bit is set, is undefined.

This instruction is not available for coprocessor 0, the System Control coprocessor, and the opcode may be used for other instructions.

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32- bit processors

vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) $\neq 0^2$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← COP_SW (z, rt)
StoreMemory (uncached, WORD, dataword, pAddr, vAddr, DATA)

Operation: 64-bit processors

$vAddr \leftarrow$ sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) $\neq 0^2$ then SignalException(AddressError) endif
(pAddr, uncached) $\leftarrow$ AddressTranslation (vAddr, DATA, STORE)
pAddr $\leftarrow$ $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || 0^2)
byte $\leftarrow$ $vAddr_{2..0}$ xor (BigEndianCPU || 0^2)
dataword$\leftarrow$ COP_SW (z, rt)
datadouble $\leftarrow$ $0^{32-8*byte}$ || dataword || 0^{8*byte}
StoreMemory (uncached, WORD, datadouble, pAddr, vAddr DATA)

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error
Reserved Instruction
Coprocessor Unusable

Store Word Left SWL

31 26	25 21	20 16	15 0
SWL 1 0 1 0 1 0	base	rt	offset
6	5	5	16

Format: SWL rt, offset(base) **MIPS I**

Purpose: To store the most-significant part of a word to an unaligned memory address.

Description: memory[base+offset] ← rt

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of four consecutive bytes forming a word in memory (*W*) starting at an arbitrary byte boundary. A part of *W*, the most-significant one to four bytes, is in the aligned word containing *EffAddr*. The same number of the most-significant (left) bytes from the word in GPR *rt* are stored into these bytes of *W*.

If GPR *rt* is a 64-bit register, the source word is the low word of the register.

Figures A-4 illustrates this operation for big-endian byte ordering for 32-bit and 64-bit registers. The four consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, two bytes, is contained in the aligned word containing the most-significant byte at 2. First, SWL stores the most-significant two bytes of the low-word from the source register into these two bytes in memory. Next, the complementary SWR stores the remainder of the unaligned word.

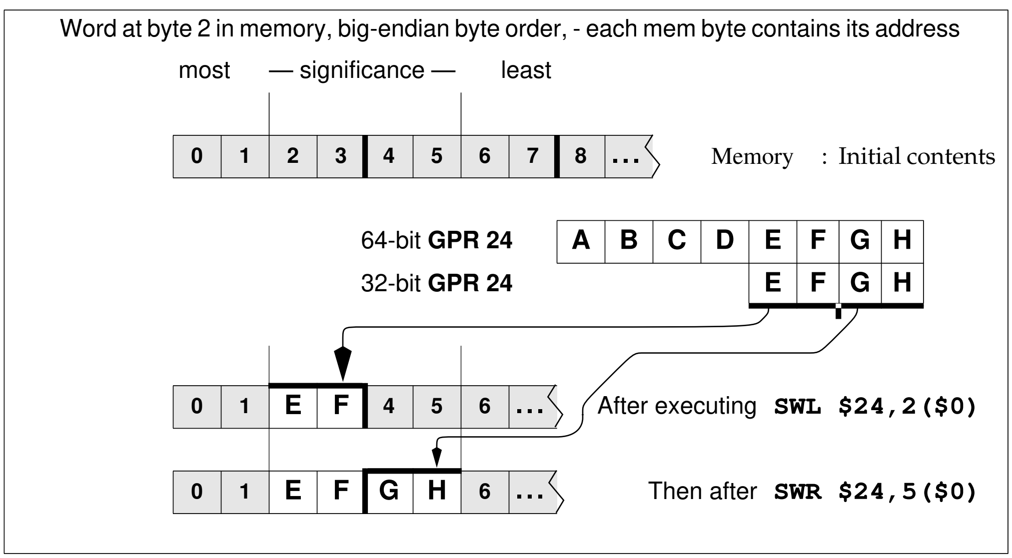


Figure A-8 Unaligned Word Store using SWL and SWR.

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word, i.e. the low two bits of the address ($vAddr_{1..0}$), and the current byte ordering mode of the processor (big- or little-endian). The table below shows the bytes stored for every combination of offset and byte ordering.

Table A-35 Bytes Stored by SWL Instruction

Memory contents and byte offsets

0	1	2	3	← big-endian
i	j	k	l	offset ($vAddr_{1..0}$)
3	2	1	0	← little-endian
most			least	

— significance —

Initial contents of Dest Register

64-bit register

A	B	C	D	E	F	G	H

most — significance — least

32-bit register

E	F	G	H

Memory contents after instruction (shaded is unchanged)

Big-endian byte ordering				$vAddr_{1..0}$	Little-endian byte ordering			
E	F	G	H	0	i	j	k	E
i	E	F	G	1	i	j	E	F
i	j	E	F	2	i	E	F	G
i	j	k	E	3	E	F	G	H

Operation: 32-bit Processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{(PSIZE-1)..2}$ || ($pAddr_{1..0}$ xor $ReverseEndian^2$)
If BigEndianMem = 0 then
    pAddr ← $pAddr_{(PSIZE-1)..2}$ || 0^2
endif
byte ← $vAddr_{1..0}$ xor $BigEndianCPU^2$
dataword ← $0^{24-8*byte}$ || $GPR[rt]_{31..24-8*byte}$
StoreMemory (uncached, byte, dataword, pAddr, vAddr, DATA)

Operation: 64-bit Processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{(PSIZE-1)..3}$ || ($pAddr_{2..0}$ xor $ReverseEndian^3$)
If BigEndianMem = 0 then
    pAddr ← $pAddr_{(PSIZE-1)..2}$ || 0^2
endif
byte ← $vAddr_{1..0}$ xor $BigEndianCPU^2$
if ($vAddr_2$ xor BigEndianCPU) = 0 then
    datadouble ← 0^{32} || $0^{24-8*byte}$ || $GPR[rt]_{31..24-8*byte}$
else
    datadouble ← $0^{24-8*byte}$ || $GPR[rt]_{31..24-8*byte}$ || 0^{32}
endif
StoreMemory(uncached, byte, datadouble, pAddr, vAddr, DATA)

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Bus Error
Address Error

SWR

31 26	25 21	20 16	15 0
SWR 101110	base	rt	offset
6	5	5	16

Format: SWR rt, offset(base) **MIPS I**

Purpose: To store the least-significant part of a word to an unaligned memory address.

Description: memory[base+offset] ← rt

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr)*. *EffAddr* is the address of the least-significant of four consecutive bytes forming a word in memory *(W)* starting at an arbitrary byte boundary. A part of *W*, the least-significant one to four bytes, is in the aligned word containing *EffAddr*. The same number of the least-significant (right) bytes from the word in GPR *rt* are stored into these bytes of *W*.

If GPR *rt* is a 64-bit register, the source word is the low word of the register.

Figures A-4 illustrates this operation for big-endian byte ordering for 32-bit and 64-bit registers. The four consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, two bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR stores the least-significant two bytes of the low-word from the source register into these two bytes in memory. Next, the complementary SWL stores the remainder of the unaligned word.

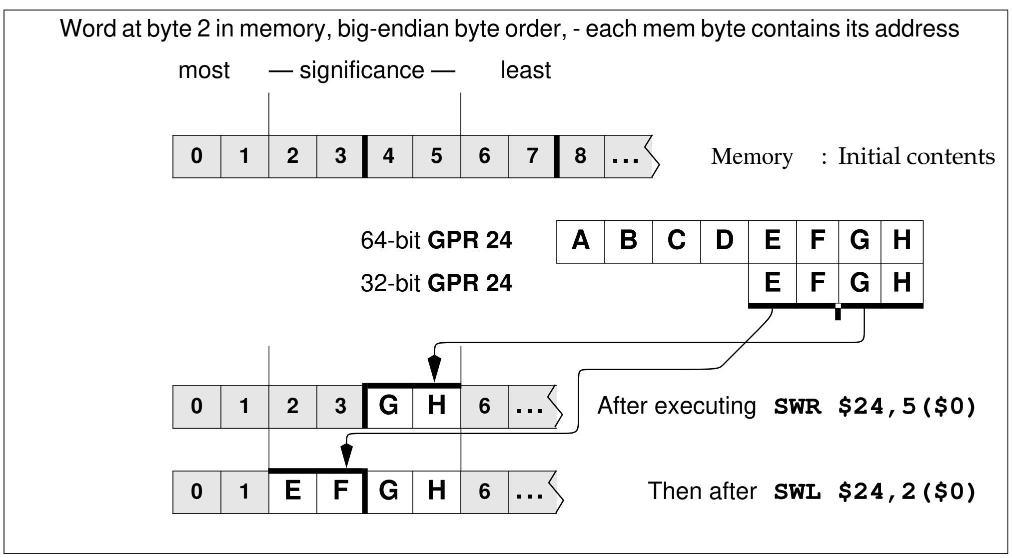


Figure A-9 Unaligned Word Store using SWR and SWL.

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word, i.e. the low two bits of the address ($vAddr_{1..0}$), and the current byte ordering mode of the processor (big- or little-endian). The tabel below shows the bytes stored for every combination of offset and byte ordering.

Table A-36 Bytes Stored by SWR Instruction

Memory contents and byte offsets

0	1	2	3	← big-endian
i	j	k	l	offset ($vAddr_{1..0}$)
3	2	1	0	← little-endian
most			least	

— significance —

Initial contents of Dest Register

64-bit register

A	B	C	D	E	F	G	H
most			— significance —				least

32-bit register	E	F	G	H

Memory contents after instruction (shaded is unchanged)

Big-endian byte ordering				$vAddr_{1..0}$	Little-endian byte ordering			
H	j	k	l	0	**E**	**F**	**G**	**H**
G	**H**	k	l	1	**F**	**G**	**H**	l
F	**G**	**H**	l	2	**G**	**H**	k	l
E	**F**	**G**	**H**	3	**H**	j	k	l

Restrictions:

None

Operation: 32- bit Processors

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{(PSIZE-1)..2}$ || ($pAddr_{1..0}$ xor $ReverseEndian^2$)
BigEndianMem = 0 then
    pAddr ← $pAddr_{(PSIZE-1)..2}$ || 0^2
endif
byte ← $vAddr_{1..0}$ xor $BigEndianCPU^2$
dataword ← $GPR[rt]_{31-8*byte}$ || 0^{8*byte}
StoreMemory (uncached, WORD-byte, dataword, pAddr, vAddr, DATA)

Operation: 64-bit Processors

$vAddr \leftarrow$ sign_extend(offset) + GPR[base]
(pAddr, uncached) $\leftarrow$ AddressTranslation (vAddr, DATA, STORE)
$pAddr \leftarrow pAddr_{(PSIZE-1)..3} \mid\mid (pAddr_{2..0}$ xor $ReverseEndian^3)$
If BigEndianMem = 0 then
    $pAddr \leftarrow pAddr_{(PSIZE-1)..2} \mid\mid 0^2$
endif
$byte \leftarrow vAddr_{1..0}$ xor $BigEndianCPU^2$
if ($vAddr_2$ xor BigEndianCPU) = 0 then
    $datadouble \leftarrow 0^{32} \mid\mid GPR[rt]_{31-8*byte..0} \mid\mid 0^{8*byte}$
else
    $datadouble \leftarrow GPR[rt]_{31-8*byte..0} \mid\mid 0^{8*byte} \mid\mid 0^{32}$
endif
StoreMemory(uncached, WORD-byte, datadouble, pAddr, vAddr, DATA)

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Bus Error
Address Error

31 26	25 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0000 0000 0000 0	stype	SYNC 0 0 1 1 1 1
6	15	5	6

Format: SYNC (stype = 0 implied) **MIPS II**

Purpose: To order loads and stores to shared memory in a multiprocessor system.

Description:

To serve a broad audience, two descriptions are given. A simple description of SYNC that appeals to intuition is followed by a precise and detailed description.

A Simple Description:

SYNC affects only uncached and cached coherent loads and stores. The loads and stores that occur prior to the SYNC must be completed before the loads and stores after the SYNC are allowed to start.

Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.

A Precise Description:

If the *stype* field has a value of zero, every synchronizable load and store that occurs in the instruction stream prior to the SYNC instruction must be globally performed before any synchronizable load or store that occurs after the SYNC may be performed with respect to any other processor or coherent I/O module.

Sync does not guarantee the order in which instruction fetches are performed.

The *stype* values 1-31 are reserved; they produce the same result as the value zero.

Synchronizable: A load or store instruction is *synchronizable* if the load or store occurs to a physical location in shared memory using a virtual location with a memory access type of either uncached or cached coherent. *Shared memory* is memory that can be accessed by more than one processor or by a coherent I/O system module.

Memory Access Types on page A-12 contains information on memory access types.

Performed load: A load instruction is *performed* when the value returned by the load has been determined. The result of a load on processor A has been *determined* with respect to processor or coherent I/O module B when a subsequent store to the location by B cannot affect the value returned by the load. The store by B must use the same memory access type as the load.

Performed store: A store instruction is *performed* when the store is observable. A store on processor A is *observable* with respect to processor or coherent I/O module B when a subsequent load of the location by B returns the value written by the store. The load by B must use the same memory access type as the store.

Globally performed load: A load instruction is *globally performed* when it is performed with respect to all processors and coherent I/O modules capable of storing to the location.

Globally performed store: A store instruction is *globally performed* when it is globally observable. It is *globally observable* when it observable by all processors and I/O modules capable of loading from the location.

Coherent I/O module: A *coherent I/O module* is an Input/Output system component that performs coherent Direct Memory Access (DMA). It reads and writes memory independently as though it were a processor doing loads and stores to locations with a memory access type of cached coherent.

Restrictions:

The effect of SYNC on the global order of the effects of loads and stores for memory access types other than uncached and cached coherent is not defined.

Operation:

```
SyncOperation(stype)
```

Exceptions:

Reserved Instruction

Programming Notes:

A processor executing load and store instructions observes the effects of the loads and stores that use the same memory access type in the order that they occur in the instruction stream; this is known as *program order*. A *parallel program* has multiple instruction streams that can execute at the same time on different processors. In multiprocessor (MP) systems, the order in which the effects of loads and stores are observed by other processors, the *global order* of the loads and stores, determines the actions necessary to reliably share data in parallel programs.

When all processors observe the effects of loads and stores in program order, the system is *strongly ordered*. On such systems, parallel programs can reliably share data without explicit actions in the programs. For such a system, SYNC has the same effect as a NOP. Executing SYNC on such a system is not necessary, but is also not an error.

If a multiprocessor system is not strongly ordered, the effects of load and store instructions executed by one processor may be observed out of program order by other processors. On such systems, parallel programs must take explicit actions in order to reliably share data. At critical points in the program, the effects of loads and stores from an instruction stream must occur in the same order for all processors. SYNC separates the loads and stores executed on the processor into two groups and the effects of these groups are seen in program order by all processors. The effect of all loads and stores in one group is seen by all processors before the effect of any load or store in the other group. In effect, SYNC causes the system to be strongly ordered for the executing processor at the instant that the SYNC is executed.

Many MIPS-based multiprocessor systems are strongly ordered or have a mode in which they operate as strongly ordered for at least one memory access type. The MIPS architecture also permits MP systems that are not strongly ordered. SYNC enables the reliable use of shared memory on such systems. A parallel program that does not use SYNC will generally not operate on a system that is not strongly ordered, however a program that does use SYNC will work on both types of systems. System-specific documentation will describe the actions necessary to reliably share data in parallel programs for that system.

The behavior of a load or store using one memory access type is undefined if a load or store was previously made to the same physical location using a different memory access type. The presence of a SYNC between the references does not alter this behavior. See page A-13 for a more complete discussion.

SYNC affects the order in which the effects of load and store instructions appears to all processors; it not generally affect the physical memory-system ordering or synchronization issues that arise in system programming. The effect of SYNC on implementation specific aspects of the cached memory system, such as writeback buffers, is not defined. The effect of SYNC on reads or writes to memory caused by privileged implementation-specific instructions, such as CACHE, is not defined.

Prefetch operations have no effects detectable by user-mode programs so ordering the effects of prefetch operations is not meaningful.

EXAMPLE : These code fragments show how SYNC can be used to coordinate the use of shared data between separate writer and reader instruction streams in a multiprocessor environment. The FLAG location is used by the instruction streams to determine whether the shared data item DATA is valid. The SYNC executed by processor A forces the store of DATA to be performed globally before the store to FLAG is performed. The SYNC executed by processor B ensures that DATA is not read until after the FLAG value indicates that the shared data is valid.

Processor A (writer)		
# Conditions at entry:		
# The value 0 has been stored in FLAG and that value is observable by B.		
SW	R1, DATA	# change shared DATA value
LI	R2, 1	
SYNC		# perform DATA store before performing FLAG store
SW	R2, FLAG	# say that the shared DATA value is valid

Processor B (reader)			
	LI	R2, 1	
1:	LW	R1, FLAG	# get FLAG
	BNE	R2, R1, 1B	# if it says that DATA is not valid, poll again
	NOP		
	SYNC		# FLAG value checked before doing DATA reads
	LW	R1, DATA	# read (valid) shared DATA values

Implementation Notes:

There may be side effects of uncached loads and stores that affect cached coherent load and store operations. To permit the reliable use of such side effects, buffered uncached stores that occur before the SYNC must be written to memory before cached coherent loads and stores after the SYNC may be performed.

System Call SYSCALL

31 26	25 6	5 0
SPECIAL 0 0 0 0 0 0	Code	SYSCALL 0 0 1 1 0 0
6	20	6

Format: SYSCALL **MIPS I**

Purpose: To cause a System Call exception.

Description:

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Restrictions:

None

Operation:

SignalException(SystemCall)

Exceptions:

System Call

TEQ

Trap if Equal

31–26	25–21	20–16	15–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TEQ 1 1 0 1 0 0
6	5	5	10	6

Format: TEQ rs, rt **MIPS II**

Purpose: To compare GPRs and do a conditional Trap.

Description: if (rs = rt) then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is equal to GPR *rt* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

```
 if GPR[rs] = GPR[rt] then
     SignalException(Trap)
endif
```

Exceptions:

Reserved Instruction
Trap

Trap if Equal Immediate **TEQI**

31–26	25–21	20–16	15–0
REGIMM 0 0 0 0 0 1	rs	TEQI 0 1 1 0 0	immediate
6	5	5	16

Format: TEQI rs, immediate **MIPS II**

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs = immediate) then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is equal to *immediate* then take a Trap exception.

Restrictions:

None

Operation:

```
if GPR[rs] = sign_extend(immediate) then
    SignalException(Trap)
endif
```

Exceptions:

Reserved Instruction
Trap

TGE

Trap if Greater or Equal

31 — 26	25 — 21	20 — 16	15 — 6	5 — 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TGE 1 1 0 0 0 0
6	5	5	10	6

Format: TGE rs, rt **MIPS II**

Purpose: To compare GPRs and do a conditional Trap.

Description: if (rs ≥ rt) then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is greater than or equal to GPR *rt* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

```
if GPR[rs] ≥ GPR[rt] then
    SignalException(Trap)
endif
```

Exceptions:

Reserved Instruction
Trap

Trap if Greater or Equal Immediate TGEI

31–26	25–21	20–16	15–0
REGIMM 0 0 0 0 0 1	rs	TGEI 0 1 0 0 0	immediate
6	5	5	16

Format: TGEI rs, immediate **MIPS II**

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs ≥ immediate) then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is greater than or equal to *immediate* then take a Trap exception.

Restrictions:

None

Operation:

```
if GPR[rs] ≥ sign_extend(immediate) then
    SignalException(Trap)
endif
```

Exceptions:

Reserved Instruction
Trap

TGEIU

Trap If Greater Or Equal Immediate Unsigned

31–26	25–21	20–16	15–0
REGIMM 0 0 0 0 0 1	rs	TGEIU 0 1 0 0 1	immediate
6	5	5	16

Format: TGEIU rs, immediate **MIPS II**

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs ≥ immediate) then Trap

Compare the contents of GPR *rs* and the 16-bit sign-extended *immediate* as unsigned integers; if GPR *rs* is greater than or equal to *immediate* then take a Trap exception.

Because the 16-bit *immediate* is sign-extended before comparison, the instruction is able to represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

Restrictions:

None

Operation:

```
if (0 || GPR[rs]) ≥ (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

Exceptions:

Reserved Instruction
Trap

31–26	25–21	20–16	15–6	5–0
SPECIAL 000000	rs	rt	code	TGEU 110001
6	5	5	10	6

Format: TGEU rs, rt **MIPS II**

Purpose: To compare GPRs and do a conditional Trap.

Description: if (rs ≥ rt) then Trap

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; if GPR *rs* is greater than or equal to GPR *rt* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

```
if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
    SignalException(Trap)
endif
```

Exceptions:

Reserved Instruction
Trap

TLT

Trap if Less Than

31–26	25–21	20–16	15–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TLT 1 1 0 0 1 0
6	5	5	10	6

Format: TLT rs, rt **MIPS II**

Purpose: To compare GPRs and do a conditional Trap.

Description: if (rs < rt) then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is less than GPR *rt* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

```
if GPR[rs] < GPR[rt] then
    SignalException(Trap)
endif
```

Exceptions:

Reserved Instruction
Trap

Trap if Less Than Immediate **TLTI**

31–26	25–21	20–16	15–0
REGIMM 000001	rs	TLTI 01010	immediate
6	5	5	16

Format: TLTI rs, immediate **MIPS II**

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs < immediate) then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is less than *immediate* then take a Trap exception.

Restrictions:

None

Operation:

```
if GPR[rs] < sign_extend(immediate) then
    SignalException(Trap)
endif
```

Exceptions:

Reserved Instruction
Trap

TLTIU — Trap if Less Than Immediate Unsigned

31–26	25–21	20–16	15–0
REGIMM 000001	rs	TLTIU 01011	immediate
6	5	5	16

Format: TLTIU rs, immediate **MIPS II**

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs < immediate) then Trap

Compare the contents of GPR *rs* and the 16-bit sign-extended *immediate* as unsigned integers; if GPR *rs* is less than *immediate* then take a Trap exception.

Because the 16-bit *immediate* is sign-extended before comparison, the instruction is able to represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

Restrictions:

None

Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

Exceptions:

Reserved Instruction
Trap

Trap if Less Than Unsigned **TLTU**

31–26	25–21	20–16	15–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TLTU 1 1 0 0 1 1
6	5	5	10	6

Format: TLTU rs, rt **MIPS II**

Purpose: To compare GPRs and do a conditional Trap.

Description: if (rs < rt) then Trap

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; if GPR *rs* is less than GPR *rt* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
    SignalException(Trap)
endif
```

Exceptions:

Reserved Instruction
Trap

TNE

Trap if Not Equal

31–26	25–21	20–16	15–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TNE 1 1 0 1 1 0
6	5	5	10	6

Format: TNE rs, rt **MIPS II**

Purpose: To compare GPRs and do a conditional Trap.

Description: if (rs ≠ rt) then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is not equal to GPR *rt* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

```
if GPR[rs] ≠ GPR[rt] then
    SignalException(Trap)
endif
```

Exceptions:

Reserved Instruction
Trap

Trap if Not Equal Immediate **TNEI**

31–26	25–21	20–16	15–0
REGIMM 0 0 0 0 0 1	rs	TNEI 0 1 1 1 0	immediate
6	5	5	16

Format: TNEI rs, immediate **MIPS II**

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs ≠ immediate) then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is not equal to *immediate* then take a Trap exception.

Restrictions:

None

Operation:

```
if GPR[rs] ≠ sign_extend(immediate) then
    SignalException(Trap)
endif
```

Exceptions:

Reserved Instruction
Trap

XOR

Exclusive OR

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	XOR 1 0 0 1 1 0
6	5	5	5	5	6

Format: XOR rd, rs, rt **MIPS I**

Purpose: To do a bitwise logical EXCLUSIVE OR.

Description: rd ← rs XOR rt

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical exclusive OR operation and place the result into GPR *rd.*

Restrictions:

None

Operation:

GPR[rd] ← GPR[rs] xor GPR[rt]

Exceptions:

None

Exclusive OR Immediate XORI

31 — 26	25 — 21	20 — 16	15 — 0
XORI 0 0 1 1 1 0	rs	rt	immediate
6	5	5	16

Format: XORI rt, rs, immediate **MIPS I**

Purpose: To do a bitwise logical EXCLUSIVE OR with a constant.

Description: rt ← rs XOR immediate

Combine the contents of GPR *rs* and the 16-bit zero-extended *immediate* in a bitwise logical exclusive OR operation and place the result into GPR *rt*.

Restrictions:

None

Operation:

```
GPR[rt] ← GPR[rs] xor zero_extend(immediate)
```

Exceptions:

None

A 7 CPU Instruction Formats

A CPU instruction is a single 32-bit aligned word. The major instruction formats are shown in Figure A-10.

I-Type (Immediate).

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

J-Type (Jump).

31 26	25 0
opcode	instr_index
6	26

R-Type (Register).

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

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

Figure A-10 CPU Instruction Formats

A 8 CPU Instruction Encoding

This section describes the encoding of user-level, i.e. non-privileged, CPU instructions for the four levels of the MIPS architecture, MIPS I through MIPS IV. Each architecture level includes the instructions in the previous level;† MIPS IV includes all instructions in MIPS I, MIPS II, and MIPS III. This section presents eight different views of the instruction encoding.

- Separate encoding tables for each architecture level.
- A MIPS IV encoding table showing the architecture level at which each opcode was originally defined and subsequently modified (if modified).
- Separate encoding tables for each architecture revision showing the changes made during that revision.

A 8.1 Instruction Decode

Instruction field names are printed in **bold** in this section.

The primary **opcode** field is decoded first. Most **opcode** values completely specify an instruction that has an immediate value or offset. **Opcode** values that do not specify an instruction specify an instruction class. Instructions within a class are further specified by values in other fields. The **opcode** values *SPECIAL* and *REGIMM* specify instruction classes. The *COP0*, *COP1*, *COP2*, *COP3*, and *COP1X* instruction classes are not CPU instructions; they are discussed in section A 8.3.

A 8.1.1 *SPECIAL* Instruction Class

The **opcode** =*SPECIAL* instruction class encodes 3-register computational instructions, jump register, and some special purpose instructions. The class is further decoded by examining the **format** field. The **format** values fully specify the CPU instructions; the *MOVCI* instruction class is not a CPU instruction class.

A 8.1.2 *REGIMM* Instruction Class

The **opcode** =*REGIMM* instruction class encodes conditional branch and trap immediate instructions. The class is further decode, and the instructions fully specified, by examining the **rt** field.

A 8.2 Instruction Subsets of MIPS III and MIPS IV Processors.

MIPS III processors, such as the R4000, R4200, R4300, R4400, and R4600, have a processor mode in which only the MIPS II instructions are valid. The MIPS II encoding table describes the MIPS II-only mode except that the Coprocessor 3 instructions (COP3, LWC3, SWC3, LDC3, SDC3) are not available and cause a Reserved Instruction exception.

† An exception to this rule is that the reserved, but never implemented, Coprocessor 3 instructions were removed or changed to another use starting in MIPS III.

MIPS IV processors, such as the R8000 and R10000, have processor modes in which only the MIPS II or MIPS III instructions are valid. The MIPS II encoding table describes the MIPS II-only mode except that the Coprocessor 3 instructions (COP3, LWC3, SWC3, LDC3, SDC3) are not available and cause a Reserved Instruction exception. The MIPS III encoding table describes the MIPS III-only mode.

A 8.3 Non-CPU Instructions in the Tables

The encoding tables show all values for the field they describe and by doing this they include some entries that are not user-level CPU instructions. The primary opcode table includes coprocessor instruction classes (COP0, COP1, COP2, COP3/COP1X) and coprocessor load/store instructions (LWCx, SWCx, LDCx, SDCx for x=1, 2, or 3). The opcode =*SPECIAL* + function =*MOVCI* instruction class is an FPU instruction.

A 8.3.1 Coprocessor 0 - ***COP0***

COP0 encodes privileged instructions for Coprocessor 0, the System Control Coprocessor. The definition of the System Control Coprocessor is processor-specific and further information on these instructions are not included in this document.

A 8.3.2 Coprocessor 1 - ***COP1***, ***COP1X***, ***MOVCI***, and CP1 load/store.

Coprocessor 1 is the floating-point unit in the MIPS architecture. *COP1*, *COP1X*, and the (opcode =*SPECIAL* + function =*MOVCI*) instruction classes encode floating-point instructions. LWC1, SWC1, LDC1, and SDC1 are floating-point loads and stores. The FPU instruction encoding is documented in section B.12.

A 8.3.3 Coprocessor 2 - ***COP2*** and CP2 load/store.

Coprocessor 2 is optional and implementation-specific. No standard processor from MIPS has implemented coprocessor 2, but MIPS' semiconductor licensees may have implemented it in a product based on one of the standard MIPS processors. At this time the standard processors are: R2000, R3000, R4000, R4200, R4300, R4400, R4600, R6000, R8000, and R10000.

A 8.3.4 Coprocessor 3 - ***COP3*** and CP3 load/store.

Coprocessor 3 is optional and implementation-specific in the MIPS I and MIPS II architecture levels. It was removed from MIPS III and later architecture levels. Note that in MIPS IV the *COP3* primary opcode was reused for the *COP1X* instruction class. No standard processor from MIPS has implemented coprocessor 2, but MIPS' semiconductor licensees may have implemented it in a product based on one of the standard MIPS processors. At this time the standard processors are: R2000, R3000, R4000, R4200, R4300, R4400, R4600, R6000, R8000, and R10000.

Table A-37 CPU Instruction Encoding - MIPS I Architecture

31 26	0
opcode	

Instructions encoded by opcode field.

opcode bits 31..29		bits 28..26: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0* δ,π	*COP1* δ,π	*COP2* δ,π	*COP3* δ,π,κ	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	*
5	101	SB	SH	SWL	SW	*	*	SWR	*
6	110	*	LWC1 π	LWC2 π	LWC3 π,κ	*	*	*	*
7	111	*	SWC1 π	SWC2 π	SWC3 π,κ	*	*	*	*

31 26		5 0
opcode = *SPECIAL*		**function**

Instructions encoded by function field when opcode field = SPECIAL.

function bits 5..3		bits 2..0: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	001	JR	JALR	*	*	SYSCALL	BREAK	*	*
2	010	MFHI	MTHI	MFLO	MTLO	*	*	*	*
3	011	MULT	MULTU	DIV	DIVU	*	*	*	*
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

31 26		20 16	0
opcode = *REGIMM*		**rt**	

Instructions encoded by the rt field when opcode field = REGIMM.

rt bits 20..19		bits 18..16: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	00	BLTZ	BGEZ	=	=	=	=	=	=
1	01	=	=	=	=	=	=	=	=
2	10	BLTZAL	BGEZAL	=	=	=	=	=	=
3	11	=	=	=	=	=	=	=	=

Table A-38 CPU Instruction Encoding - MIPS II Architecture

31..26	25..0
opcode	

Instructions encoded by opcode field.

opcode bits 31..29	bits 28..26	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0* δ,π	*COP1* δ,π	*COP2* δ,π	*COP3* δ,π,κ	BEQL	BNEL	BLEZL	BGTZL
3	011	*	*	*	*	*	*	*	*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	*
5	101	SB	SH	SWL	SW	*	*	SWR	ρ
6	110	LL	LWC1 π	LWC2 π	LWC3 π,κ	*	LDC1 π	LDC2 π	LDC3 π,κ
7	111	SC	SWC1 π	SWC2 π	SWC3 π,κ	*	SDC1 π	SDC2 π	SDC3 π,κ

31..26		5..0
opcode = *SPECIAL*		**function**

Instructions encoded by function field when opcode field = SPECIAL.

function bits 5..3	bits 2..0	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	001	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	*	*	*	*
3	011	MULT	MULTU	DIV	DIVU	*	*	*	*
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	*	*	*	*
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	*	*	*	*	*	*	*	*

31..26		20..16	..0
opcode = *REGIMM*		**rt**	

Instructions encoded by the rt field when opcode field = REGIMM.

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

Table A-39 CPU Instruction Encoding - MIPS III Architecture

31..26	25..0
opcode	

Instructions encoded by opcode field.

opcode bits 31..29 \ bits 28..26		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0* δ,π	*COP1* δ,π	*COP2* δ,π	*	BEQL	BNEL	BLEZL	BGTZL
3	011	DADDI	DADDIU	LDL	LDR	*	*	*	*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU
5	101	SB	SH	SWL	SW	SDL	SDR	SWR	ρ
6	110	LL	LWC1 π	LWC2 π	*	LLD	LDC1 π	LDC2 π	LD
7	111	SC	SWC1 π	SWC2 π	*	SCD	SDC1 π	SDC2 π	SD

31..26		5..0
opcode = *SPECIAL*		**function**

Instructions encoded by function field when opcode field = SPECIAL.

function bits 5..3 \ bits 2..0		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	001	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	DSLLV	*	DSRLV	DSRAV
3	011	MULT	MULTU	DIV	DIVU	DMULT	DMULTU	DDIV	DDIVU
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	DADD	DADDU	DSUB	DSUBU
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	DSLL	*	DSRL	DSRA	DSLL32	*	DSRL32	DSRA32

31..26		20..16	..0
opcode = *REGIMM*		**rt**	

Instructions encoded by the rt field when opcode field = REGIMM.

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

Table A-40 CPU Instruction Encoding - MIPS IV Architecture

31..26	25..0
opcode	

Instructions encoded by opcode field.

opcode bits 31..29	bits 28..26 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1 001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2 010	*COP0* δ,π	*COP1* δ,π	*COP2* δ,π	*COP1X* δ,π	BEQL	BNEL	BLEZL	BGTZL
3 011	DADDI	DADDIU	LDL	LDR		*	*	*
4 100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU
5 101	SB	SH	SWL	SW	SDL	SDR	SWR	ρ
6 110	LL	LWC1 π	LWC2 π	PREF	LLD	LDC1 π	LDC2 π	LD
7 111	SC	SWC1 π	SWC2 π	*	SCD	SDC1 π	SDC2 π	SD

31..26	25..6	5..0
opcode = *SPECIAL*		**function**

Instructions encoded by function field when opcode field = SPECIAL.

function bits 5..3	bits 2..0 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	SLL	*MOVCI* δ,μ	SRL	SRA	SLLV	*	SRLV	SRAV
1 001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2 010	MFHI	MTHI	MFLO	MTLO	DSLLV	*	DSRLV	DSRAV
3 011	MULT	MULTU	DIV	DIVU	DMULT	DMULTU	DDIV	DDIVU
4 100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5 101	*	*	SLT	SLTU	DADD	DADDU	DSUB	DSUBU
6 110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7 111	DSLL	*	DSRL	DSRA	DSLL32	*	DSRL32	DSRA32

31..26	25..21	20..16	15..0
opcode = *REGIMM*		**rt**	

Instructions encoded by the rt field when opcode field = REGIMM.

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

Table A-41 Architecture Level in Which CPU Instructions are Defined or Extended.

The architecture level in which each MIPS IVencoding was defined is indicated by a subscript 1, 2, 3, or 4 (for architecture level I, II, III, or IV). If an instruction or instruction class was later extended, the extending level is indicated after the defining level.

31 26	0
opcode	

Instructions encoded by opcode field.

opcode bits 31..29	bits 28..26: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	*SPECIAL* $_{1\text{-}4}$	*REGIMM* $_{1,2}$	J $_1$	JAL $_1$	BEQ $_1$	BNE $_1$	BLEZ $_1$	BGTZ $_1$
1 001	ADDI $_1$	ADDIU $_1$	SLTI $_1$	SLTIU $_1$	ANDI $_1$	ORI $_1$	XORI $_1$	LUI $_1$
2 010	*COP0* $_1$	*COP1* $_{1,2,3,4}$	*COP2* $_1$	*COP1X* $_4$	BEQL $_2$	BNEL $_2$	BLEZL $_2$	BGTZL $_2$
3 011	DADDI $_3$	DADDIU $_3$	LDL $_3$	LDR $_3$	$*_1$	$*_1$	$*_1$	$*_1$
4 100	LB $_1$	LH $_1$	LWL $_1$	LW $_1$	LBU $_1$	LHU $_1$	LWR $_1$	LWU $_3$
5 101	SB $_1$	SH $_1$	SWL $_1$	SW $_1$	SDL $_3$	SDR $_3$	SWR $_1$	ρ_2
6 110	LL $_2$	LWC1 $_1$	LWC2 $_1$	PREF $_4$	LLD $_3$	LDC1 $_2$	LDC2 $_2$	LD $_3$
7 111	SC $_2$	SWC1 $_1$	SWC2 $_1$	$*_3$	SCD $_3$	SDC1 $_2$	SDC2 $_2$	SD $_3$

31 26		5 0
opcode = *SPECIAL*		**function**

Instructions encoded by function field when opcode field = SPECIAL.

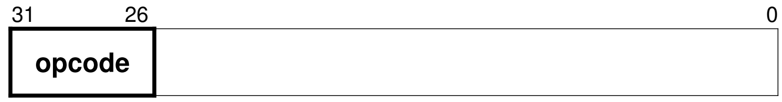
function bits 5..3	bits 2..0: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	SLL $_1$	*MOVCI* $_4$	SRL $_1$	SRA $_1$	SLLV $_1$	$*_1$	SRLV $_1$	SRAV $_1$
1 001	JR $_1$	JALR $_1$	MOVZ $_4$	MOVN $_4$	SYSCALL $_1$	BREAK $_1$	$*_1$	SYNC $_2$
2 010	MFHI $_1$	MTHI $_1$	MFLO $_1$	MTLO $_1$	DSLLV $_3$	$*_1$	DSRLV $_3$	DSRAV $_3$
3 011	MULT $_1$	MULTU $_1$	DIV $_1$	DIVU $_1$	DMULT $_3$	DMULTU $_3$	DDIV $_3$	DDIVU $_3$
4 100	ADD $_1$	ADDU $_1$	SUB $_1$	SUBU $_1$	AND $_1$	OR $_1$	XOR $_1$	NOR $_1$
5 101	$*_1$	$*_1$	SLT $_1$	SLTU $_1$	DADD $_3$	DADDU $_3$	DSUB $_3$	DSUBU $_3$
6 110	TGE $_2$	TGEU $_2$	TLT $_2$	TLTU $_2$	TEQ $_2$	$*_1$	TNE $_2$	$*_1$
7 111	DSLL $_3$	$*_1$	DSRL $_3$	DSRA $_3$	DSLL32 $_3$	$*_1$	DSRL32 $_3$	DSRA32 $_3$

31 26		20 16	0
opcode = *REGIMM*		**rt**	

Instructions encoded by the rt field when opcode field = REGIMM.

rt bits 20..19	bits 18..16: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00	BLTZ $_1$	BGEZ $_1$	BLTZL $_2$	BGEZL $_2$	$*_1$	$*_1$	$*_1$	$*_1$
1 01	TGEI $_2$	TGEIU $_2$	TLTI $_2$	TLTIU $_2$	TEQI $_2$	$*_1$	TNEI $_2$	$*_1$
2 10	BLTZAL $_1$	BGEZAL $_1$	BLTZALL $_2$	BGEZALL $_2$	$*_1$	$*_1$	$*_1$	$*_1$
3 11	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$

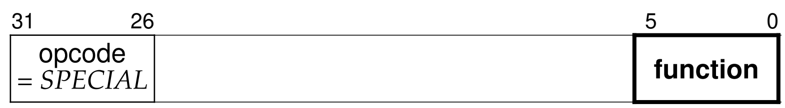
Table A-42 CPU Instruction Encoding Changes - MIPS II Revision.



An instruction encoding is shown if the instruction is added in this revision.

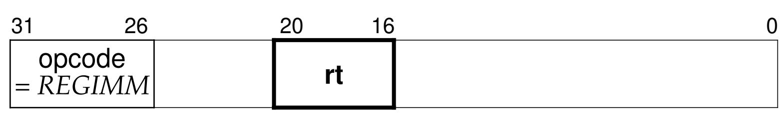
Instructions encoded by opcode field.

opcode bits 31..29 \ bits 28..26	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001								
2 010					BEQL	BNEL	BLEZL	BGTZL
3 011								
4 100								
5 101								ρ
6 110	LL					LDC1 π	LDC2 π	LDC3 π
7 111	SC					SDC1 π	SDC2 π	SDC3 π



Instructions encoded by function field when opcode field = SPECIAL.

function bits 5..3 \ bits 2..0	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001								SYNC
2 010								
3 011								
4 100								
5 101								
6 110	TGE	TGEU	TLT	TLTU	TEQ		TNE	
7 111								



Instructions encoded by the rt field when opcode field = REGIMM.

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

Table A-43 CPU Instruction Encoding Changes - MIPS III Revision.

31 26	0
opcode	

An instruction encoding is shown if the instruction is added or modified in this revision.

Instructions encoded by opcode field.

opcode bits 31..29	bits 28..26 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001								
2 010				* (was *COP3*)				
3 011	DADDI	DADDIU	LDL	LDR				
4 100								LWU
5 101					SDL	SDR		
6 110				* (was LWC3)	LLD			LD (was LDC3)
7 111				* (was SWC3)	SCD			SD (was SDC3)

31 26		5 0
opcode = *SPECIAL*		**function**

Instructions encoded by function field when opcode field = SPECIAL.

function bits 5..3	bits 2..0 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001								
2 010					DSLLV		DSRLV	DSRAV
3 011					DMULT	DMULTU	DDIV	DDIVU
4 100								
5 101					DADD	DADDU	DSUB	DSUBU
6 110								
7 111	DSLL		DSRL	DSRA	DSLL32		DSRL32	DSRA32

31 … 26	25 … 21	20 … 16	15 … 0
opcode = *REGIMM*		**rt**	

rt	bits 18..16			Instructions encoded by the rt field when opcode field = REGIMM.				
bits 20..19	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00								
1 01								
2 10								
3 11								

Table A-44 CPU Instruction Encoding Changes - MIPS IV Revision.

31 26	0
opcode	

An instruction encoding is shown if the instruction is added or modified in this revision.

opcode bits 28..26 — Instructions encoded by opcode field.

bits 31..29	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001								
2 010				*COP1X* δ,π				
3 011								
4 100								
5 101								
6 110				PREF				
7 111								

31 26		5 0
opcode = *SPECIAL*		**function**

function bits 2..0 — Instructions encoded by function field when opcode field = SPECIAL.

bits 5..3	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000		*MOVCI* δ,μ						
1 001			MOVZ	MOVN				
2 010								
3 011								
4 100								
5 101								
6 110								
7 111								

31 26		20 16	0
opcode = *REGIMM*		**rt**	

rt bits 18..16 — Instructions encoded by the rt field when opcode field = REGIMM.

bits 20..19	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00								
1 01								
2 10								
3 11								

Key to notes in CPU instruction encoding tables:

- * This opcode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception.
- = This opcode is reserved for future use. An attempt to execute it produces an undefined result. The result may be a Reserved Instruction exception but this is not guaranteed.
- δ (also *italic* opcode name) This opcode indicates an instruction class. The instruction word must be further decoded by examing additional tables that show values for another instruction field.
- π This opcode is a coprocessor operation, not a CPU operation. If the processor state does not allow access to the specified coprocessor, the instruction causes a Coprocessor Unusable exception. It is included in the table because it uses a primary opcode in the instruction encoding map.
- κ This opcode is removed in a later revision of the architecture. If a MIPS III or MIPS IV processor is operated in MIPS II-only mode this opcode will cause a Reserved Instruction exception.
- μ This opcode indicates a class of coprocessor 1 instructions. If the processor state does not allow access to coprocessor 1, the opcode causes a Coprocessor Unusable exception. It is included in the table because the encoding uses a location in what is otherwise a CPU instruction encoding map. Further encoding information for this instruction class is in the FPU Instruction Encoding tables.
- ρ This opcode is reserved for Coprocessor 0 (System Control Coprocessor) instructions that require base+offset addressing. If the instruction is used for COP0 in an implementation, an attempt to execute it without Coprocessor 0 access privilege will cause a Coprocessor Unusable exception. If the instruction is not used in an implementation, it will cause a Reserved Instruction exception.

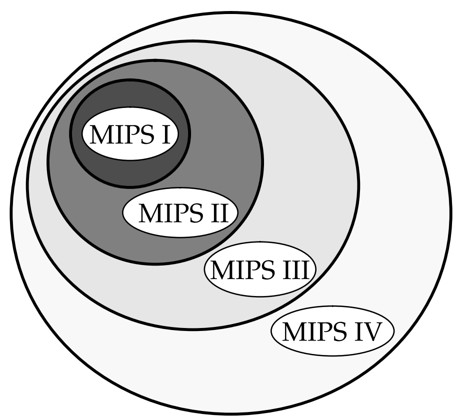
FPU Instruction Set

B

B 1 Introduction

This appendix describes the instruction set architecture (ISA) for the floating-point unit (FPU) in the MIPS IV architecture. In the MIPS architecture, the FPU is coprocessor 1, an optional processor implementing IEEE Standard 754† floating-point operations. The FPU also provides a few additional operations not defined by the IEEE standard.

The original MIPS I FPU ISA has been extended in a backward-compatible fashion three times. The ISA extensions are inclusive as the diagram illustrates; each new architecture level (or version) includes the former levels. The description of an architectural feature includes the architecture level in which the feature is (first) defined or extended. The feature is also available in all later (higher) levels of the architecture.



MIPS Architecture Extensions

† IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic"

In addition to an ISA, the architecture definition includes processing resources, such as the coprocessor general register set. The 32-bit registers in MIPS I were changed to 64-bit registers in MIPS III in a way that is not backwards compatible. For changes such as this, processors implementing higher levels of the architecture have a way to provide the processing resource model for earlier levels. For the FPU there is a mode to select the 32-bit or 64-bit register model. The practical result is that a processor implementing MIPS IV is also able to run MIPS I, MIPS II, or MIPS III binary programs without change.

If coprocessor 1 is not enabled, an attempt to execute a floating-point instruction will cause a Coprocessor Unusable exception. Enabling coprocessor 1 is a privileged operation provided by the System Control Coprocessor. Every system environment will either enable the FPU automatically or provide a means for an application to request that it be enabled.

Before the instruction set is described, there is an overview of the FPU data types, registers, and computational model. The FPU instruction set is summarized by functional group then each operation is described separately in alphabetical order. The description concludes with the FPU instruction formats and opcode encoding tables. See the CPU instruction set section titled "Description of an Instruction" for a description of the organization of the individual instruction descriptions and the notation used in them.

The architecture of the floating-point coprocessor consists of:

- Data types
- Operations
- A computational model
- Processing resources (registers)
- An instruction set

The IEEE standard defines the floating-point number data types, the basic arithmetic, comparison, and conversion operations, and a computational model.

The IEEE standard defines neither specific processing resources nor an instruction set. The MIPS architecture defines fixed-point (integer) data types, FPU register sets, control and exception mechanisms, and an instruction set. The architecture include non-IEEE FPU control operations, and arithmetic operations (multiply-add, reciprocal, and reciprocal square root) that may not supply results that match the IEEE precision rules.

B 2 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 the signed integers provided by the CPU architecture

B 2.1 Floating-point formats

There are two floating-point data types provided by the FPU.

- 32-bit Single precision floating-point (type S)
- 64-bit Double precision floating-point (type D)

The floating-point formats represents numeric values as well as other special entities:

1. Numbers of the form: $(-1)^s\ 2^E\ b_0 .\ b_1\, b_2 \ldots b_{p-1}$
   where (see Table B-1):
   - 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 precision
2. Two infinities, $+\infty$ and $-\infty$
3. Signaling non-numbers (SNaNs)
4. Quiet non-numbers (QNaNs)

Table B-1 Parameters of Floating-Point Formats

parameter	Single	Double
bits of mantissa precision, p	24	53
maximum exponent, *E_max*	+127	+1023
minimum exponent, *E_min*	-126	-1022
exponent *bias*	+127	+1023
bits in exponent field, e	8	11
representation of b_0 integer bit	hidden	hidden
bits in fraction field, f	23	52
total format width in bits	32	64

The single and double floating-point formats are composed of three fields whose size is listed in Table B-1. The layouts are pictured in the figures below.

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

31	30 23	22 0
sign	exponent	fraction
1	8	23

Figure B-1 Single-Precision Floating-Point Format (S)

63	62 … 52	51 … 0
sign	exponent	fraction
1	11	52

Figure B-2 Double-Precision Floating-Point Format (D)

Values are encoded in the formats using the unbiased exponent, fraction, and sign values shown in Table B-2. The high-order bit of the fraction field, identified as b_1, is also important for NaNs.

Table B-2 Value of Single or Double Floating-Point Format Encoding

unbiased ***E***	***f***	***s***	***b₁***	**value *v***	**type of value**
E_max + 1	≠ 0		1	SNaN	Signaling NaN
			0	QNaN	Quiet NaN
E_max +1	0	1		$-\infty$	minus infinity
		0		$+\infty$	plus infinity
E_max to *E_min*		1		$-(2^E)(1.f)$	negative normalized number
		0		$+(2^E)(1.f)$	positive normalized number
E_min -1	≠ 0	1		$-(2^{E\_min})(0.f)$	negative denormalized number
		0		$+(2^{E\_min})(0.f)$	positive denormalized number
E_min -1	0	1		- 0	negative zero
		0		+ 0	positive zero

B 2.1.1 Normalized and Denormalized Numbers

For single and double formats, 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.

B 2.1.2 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, shown in Table B-2, for +infinity (+∞), -infinity (-∞), quiet NaN (QNan), and signaling NaN (SNaN).

Infinity represents a number with magnitude too large to be represented in the format; in essence it exists to represent a magnitude overflow during a computation. A correctly signed ∞ is generated as the default result in division by zero and some cases of overflow; details are in the IEEE exception condition descriptions and Table B-4 "Default Result for IEEE Exceptions Not Trapped Precisely".

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 the ∞ 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 the description of the Invalid Operation exception for a list of these cases.

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

> NOTE: The IEEE 754 Standard states that "Whether copying a signaling NaN without a change of format signals the invalid operation exception is the implementor's option". The MIPS architecture has chosen to make the formatted operand move instructions (MOV.*fmt* MOVT.*fmt* MOVF.*fmt* MOVN.*fmt* MOVZ.*fmt*) non-arithmetic and they do not signal IEEE exceptions.

QNaNs are intended to afford retrospective diagnostic information inherited from invalid or unavailable data and results. Propagation of the diagnostic information requires that information contained in the QNaNs 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. The result QNaN is one of the operand QNaN values when possible. QNaNs do have effects similar to SNaNs on operations that do not deliver a floating-point result, specifically comparisons. See the detailed description of the floating-point compare instruction (C.*cond.fmt*) for information.

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

Table B-3 Value Supplied when a new Quiet NaN is Created

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

B 2.2 Fixed-point formats

There are two floating-point data types provided by the FPU.

- 32-bit Word fixed-point (type W)
- 64-bit Longword fixed-point (type L) (defined in MIPS III)

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

31	30 … 0
sign	int
1	31

Figure B-3 Word Fixed-Point Format (W)

63	62 … 0
sign	int
1	63

Figure B-4 Longword Fixed-Point Format (L)

B 3 Floating-Point Registers

This section describes the organization and use of the two separate coprocessor 1 (CP1) register sets. The coprocessor general registers, also called Floating General Registers (FGRs) are used to transfer binary data between the FPU and the rest of the system. The general register set is also used to hold formatted FPU operand values. There are only two control registers and they are used to identify and control the FPU.

There are separate 32-bit and 64-bit wide register models. MIPS I defines the 32-bit wide register model. MIPS III defines the 64-bit model. To support programs for earlier architecture definitions, processors providing the 64-bit MIPS III register model also provide the 32-bit wide register model as a mode selection. Selecting 32 or 64-bit register model is an implementation-specific privileged operation.

B 3.1 Organization

The CP1 register organization for 32-bit and 64-bit register models is shown in Figure B-5. The coprocessor general registers are the same width as the CPU registers. The two defined control registers are 32-bits wide.

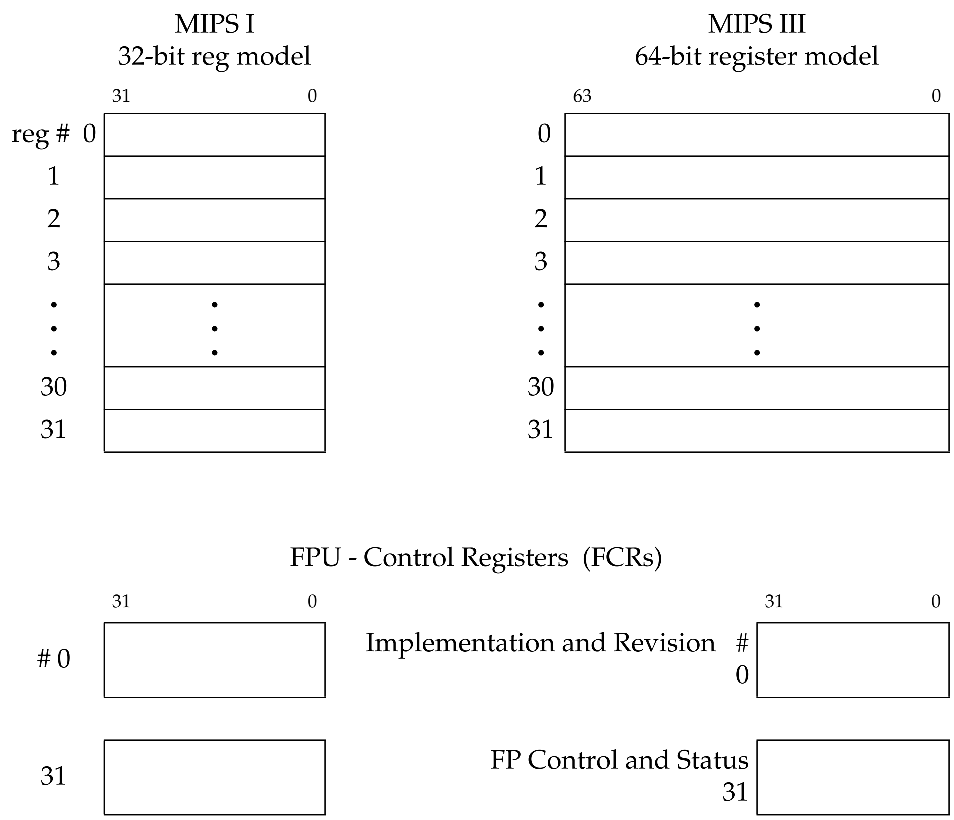


Figure B-5 Coprocessor 1 General Registers (FGRs)

B 3.2 Binary Data Transfers

The data transfer instructions move words and doublewords between the CP1 general registers and the remainder of the system. The operation of the load and move-to instructions is shown in Figure B-6 and Figure B-7. The store and move-from instructions operate in reverse, reading data from the location that the corresponding load or move-to instruction wrote it.

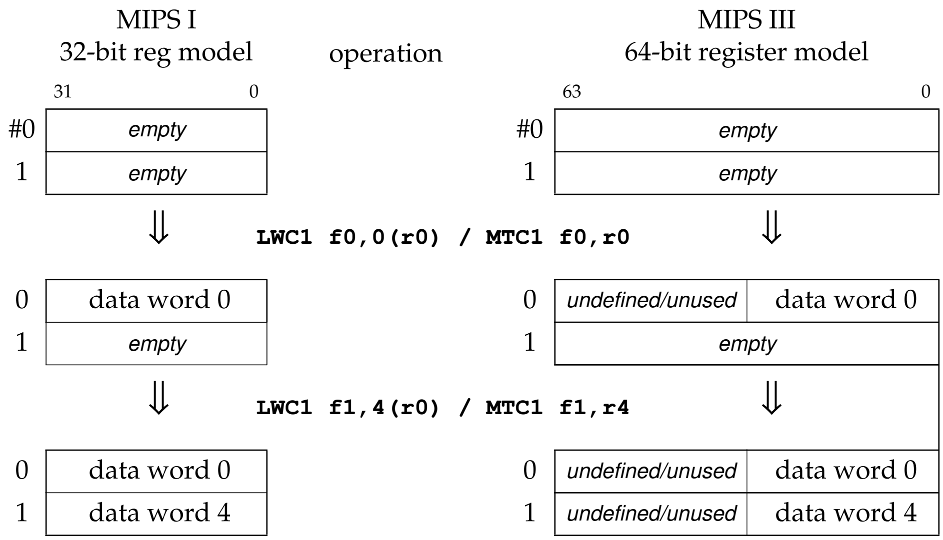
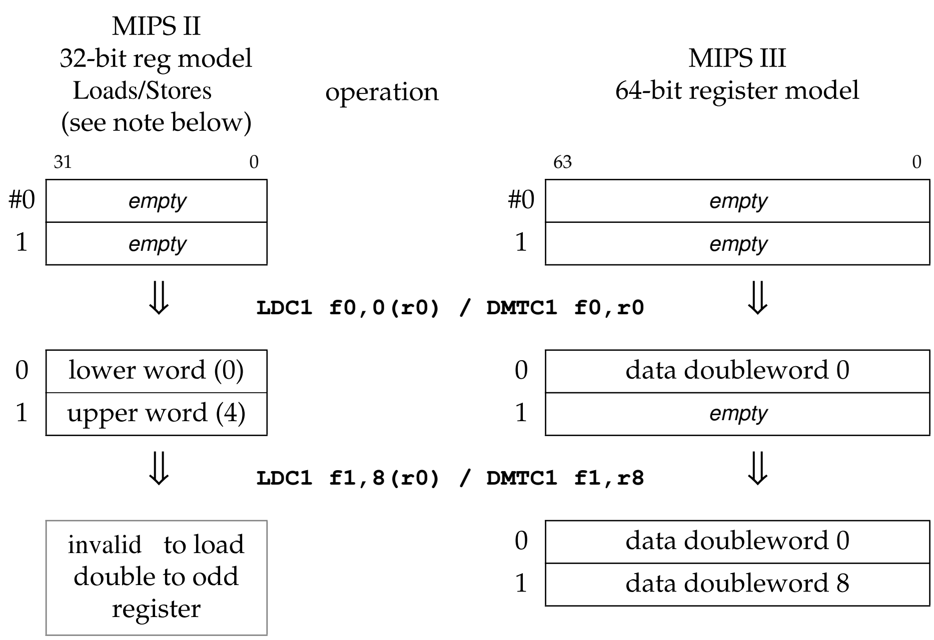


Figure B-6 Effect of FPU Word Load or Move-to Operations

Doubleword transfers to/from 32-bit registers access an aligned pair of CP1 general registers with the least-significant word of the doubleword in the lowest-numbered register.



NOTE: No 64-bit transfers are defined for the MIPS I 32-bit register model.
MIPS II defines the 64-bit loads/stores but not 64-bit moves.

Figure B-7 Effect of FPU Doubleword Load or Move-to Operations

B 3.3 Formatted Operand Layout

FPU instructions that operate on formatted operand values specify the floating-point register (FPR) that holds a value. An FPR is not necessarily the same as a CP1 general register because an FPR is 64 bits wide; if this is wider than the CP1 general registers, an aligned set of adjacent CP1 general registers is used as the FPR. The 32-bit register model provides 16 FPRs specified by the even CP1 general register numbers. The 64-bit register model provides 32 FPRs, one per CP1 general register. Operands that are only 32 bits wide (W and S formats), use only half the space in an FPR. The FPR organization and the way that operand data is stored in them is shown in the following figures. A summary of the data transfer instructions can be found in section B 6.1 on page B-19.

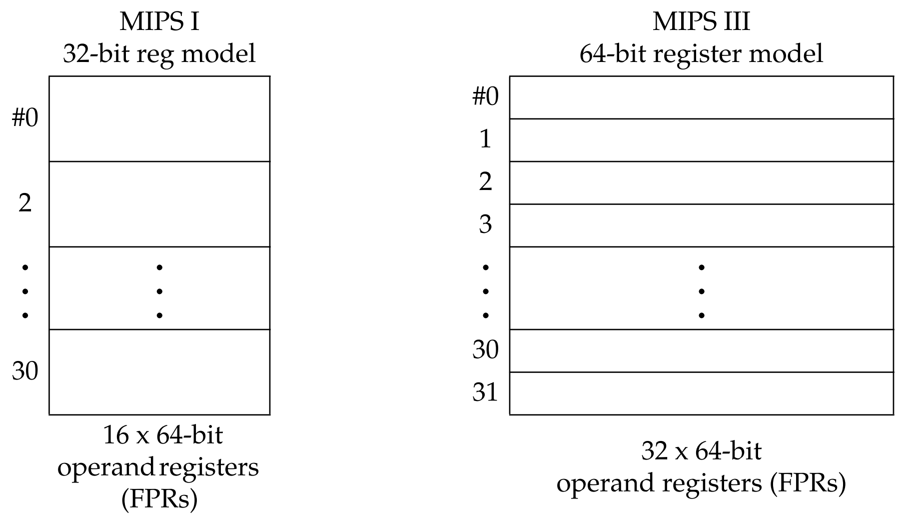


Figure B-8 Floating-point Operand Register (FPR) Organization

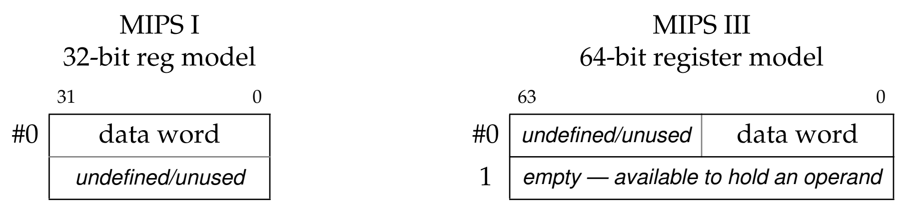


Figure B-9 Single Floating Point (S) or Word Fixed (W) Operand in an FPR

MIPS I 32-bit reg model (see note below)	31 — 0
#0	lower word
	upper word

MIPS III 64-bit register model	63 — 0
#0	data doubleword
1	*empty — available to hold an operand*

NOTE: MIPS I supports the Double floating-point (D) type; the fixed-point longword (L) operand is available starting in MIPS III

Figure B-10 Double Floating Point (D) or Long Fixed (L) Operand In an FPR

B 3.4 Implementation and Revision Register

Coprocessor control register 0 contains values that identify the implementation and revision of the FPU. Only the low-order two bytes of this register are defined as shown in Figure B-11.

32 — 16	15 — 8	7 — 0
0	Implementation	Revision
16	8	8

Figure B-11 FPU Implementation and Revision Register

The implementation field identifies a particular FPU part, but the revision number may not be relied on to reliably characterize the FPU functional version.

B 3.5 FPU Control and Status Register — FCSR

Coprocessor control register 31 Is the FPU Control and Status Register (FCSR). Access to the register is not privileged; it can be read or written by any program that can execute floating-point instructions. It controls some operations of the coprocessor and shows 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 IEEE exceptions that arose in the most recently executed instruction.
- Reports IEEE exceptions that arose, cumulatively, in completed instructions.
- Indicates the condition code result of FP compare instructions.

The contents of this register are unpredictable and undefined after a processor reset or a power-up event. Software should initialize this register.

Figure B-12 MIPS I - FPU Control and Status Register (FCSR)

31 24	23	22 18	17 12	11 7	6 2	1 0
0	c	0	cause	enables	flags	RM
8	1	5	6	5	5	2

E	V	Z	O	U	I	V	Z	O	U	I	V	Z	O	U	I
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2

Figure B-13 MIPS III - FPU Control and Status Register (FCSR)

31 25	24	23	22 18	17 12	11 7	6 2	1 0
0	FS	c	0	cause	enables	flags	RM
7	1	1	5	6	5	5	2

E	V	Z	O	U	I	V	Z	O	U	I	V	Z	O	U	I
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2

Figure B-14 MIPS IV - FPU Control and Status Register (FCSR)

31 25	24	23	22 18	17 12	11 7	6 2	1 0
FCC	FS	FCC	0	cause	enables	flags	RM
7	1	1	5	6	5	5	2

7	6	5	4	3	2	1
31	30	29	28	27	26	25

0
23

E	V	Z	O	U	I	V	Z	O	U	I	V	Z	O	U	I
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2

All fields in the FCSR are readable and writable.

FCC Floating-Point Condition Codes. These bits record the result of FP compares and are tested for FP conditional branches; the FCC bit to use is specified in the compare or branch instruction. The 0^{th} FCC bit is the same as the *c* bit in MIPS I.

FS Flush to Zero. When FS is set, denormalized results are flushed to zero instead of causing an unimplemented operation exception. When a denormalized operand value is encountered, zero may be used instead of the denorm; this is implementation specific.

c Condition Bit. This bit records the result of FP compares and is tested by FP conditional branches. In MIPS IV this becomes the 0^{th} FCC bit.

cause	Cause bits. These bits indicate the exception conditions that arise during the execution of an FPU arithmetic instruction in precise exception mode. A bit is set to 1 if the corresponding exception condition arises during the execution of an instruction and 0 otherwise. By reading the registers, the exception conditions caused by the preceding FPU arithmetic instruction can be determined. The meaning of the individual bits is: E Unimplemented Operation V Invalid Operation Z Divide by Zero O Overflow U Underflow I Inexact Result
enables	Enable bits (see cause field for bit names). These bits control, for each of the five conditions individually, whether a trap is taken when the IEEE exception condition occurs. The trap occurs when both an enable bit and the corresponding cause bit are set during an FPU arithmetic operation or by moving a value to the FCSR. The meaning of the individual bits is the same as the cause bits. Note that the “E” cause bit has no corresponding enable bit; the non-IEEE Unimplemented Operation exception defined by MIPS is always enabled.
flags	Flag bits. (see cause field for bit names) This field shows the exception conditions that have occurred for completed instructions since it was last reset. For a completed FPU arithmetic operation that raises an exception condition the corresponding bits in the flag field are set and the others are unchanged. This field is never reset by hardware and must be explicitly reset by user software.
RM	Rounding Mode. The rounding mode used for most floating-point operations (some FP instructions use a specific rounding mode). The rounding modes are: 0 RN -- Round to Nearest Round result to the nearest representable value. When two representable values are equally near, round to the value that has a least significant bit of zero (i.e. is even). 1 RZ -- Round toward Zero Round result to the value closest to and not greater in magnitude then the result. 2 RP -- Round toward Plus infinity Round result to the value closest to and not less than the result. 3 RM -- Round toward Minus infinity Round result to the value closest to and not greater than the result.

B 4 Values in FP Registers

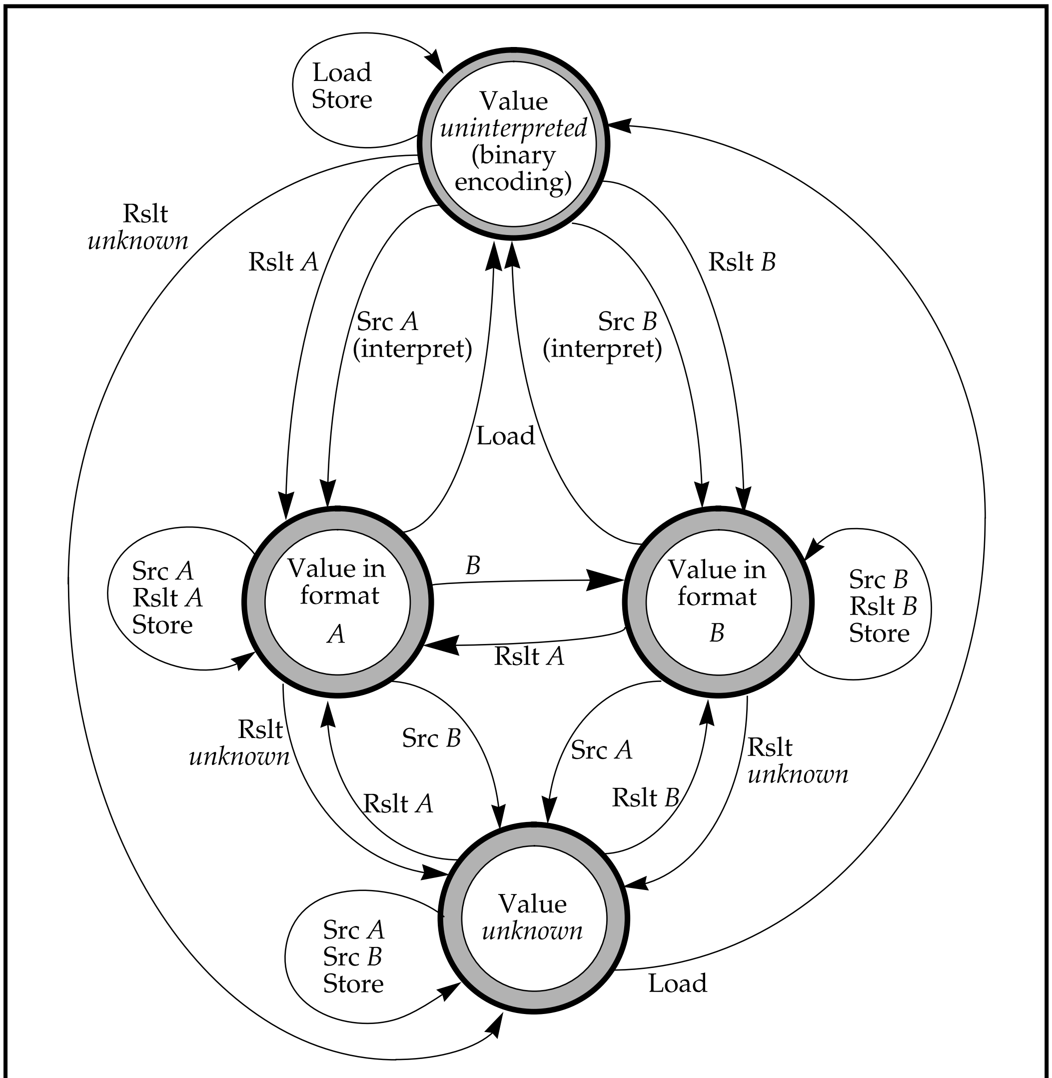
Unlike the CPU, the FPU does not interpret the binary encoding of source operands or produce a binary encoding of results for every operation. The value held in a floating-point operand register (FPR) has a format, or type and it may only be used by instructions that operate on that format. The format of a value is either *uninterpreted*, *unknown*, or one of the valid numeric formats: single and double floating-point and word and long fixed-point. The way that the formatted value in an FPR is set and changed is summarized in the state diagram in Figure B-15 and is discussed below.

The value in an FPR is always set when a value is written to the register. When a data transfer instruction writes binary data into an FPR (a load), the FPR gets a binary value that is *uninterpreted*. A computational or FP register move instruction that produces a result of type *fmt* puts a value of type *fmt* into the result register.

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

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

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



A, *B*:	Example formats
Load:	Destination of LWC1, LDC1, MTC1, or DMTC1 instructions.
Store:	Source operand of SWC1, SDC1, MFC1, or DMFC1 instructions.
Src *fmt*:	Source operand of computational instruction expecting format "*fmt*".
Rslt *fmt*:	Result of computational instruction producing value of format "*fmt*".

Figure B-15 The Effect of FPU Operations on the Format of Values Held in FPRs.

B 5 FPU Exceptions

The IEEE 754 standard specifies that:

> There are five types of exceptions that shall be signaled when detected. The signal entails setting a status flag, taking a trap, or possibly doing both. With each exception should be associated a trap under user control, ...

This function is implemented in the MIPS FPU architecture with the cause, enable, and flag fields of the control and 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.

There may be two exception modes for the FPU, precise and imprecise, and the operation of the FPU when exception conditions arise depends on the exception mode that is currently selected. Every processor is able to operate the FPU in the precise exception mode. Some processors also have an imprecise exception mode in which floating-point performance is greater. Selecting the exception mode, when there is a choice, is a privileged implementation-specific operation.

B 5.1 Precise Exception Mode

In precise exception mode, an exception (trap) caused by a floating-point operation is precise. A precise trap occurs before the instruction that causes the trap, or any following instruction, completes and writes results. If desired, the software trap handler can resume execution of the interrupted instruction stream after handling the exception.

The cause bit field reports per-instruction exception conditions. The cause bits are written during each floating-point arithmetic operation to show the exception conditions that arose during the operation. The bits are set to 1 if the corresponding exception condition arises and 0 otherwise.

A floating-point trap is generated any time both a cause bit and the 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 condition always generates a trap.

In a trap handler, the exception conditions that arose during the floating-point operation that trapped are reported in the cause field. Before returning from a floating-point interrupt or exception, or setting cause bits with a move to the FCSR, software must first clear the enabled cause bits by a move to the FCSR to prevent the trap from being retaken. User-mode programs can never observe enabled cause bits set. If this information is required in a user-mode handler, then it must be passed somewhere other than the status register.

For a floating-point operation that sets only non-enabled cause bits, no trap occurs and the default result defined by the IEEE standard is stored (see Table B-4). When a floating-point operation does not trap, the program can see the exception conditions that arose during the operation by reading the cause field.

The flag bit field is a cumulative report of IEEE exception conditions that arise during instructions that complete; instructions that trap do not update the flag bits. The flag bits are set to 1 if the corresponding IEEE exception is raised and unchanged otherwise. There is no flag bit for the MIPS Uniplemented Operation exception condition. 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.

B 5.2 Imprecise Exception Mode

In imprecise exception mode, an exception (trap) caused by an IEEE floating-point operation is imprecise (Unimplemented Operation exceptions must still be signaled precisely). An imprecise trap occurs at some point after the exception condition arises. In particular, it does not necessarily occur before the instruction that causes the exception, or following instructions, have completed and written results. The software trap handler can generally neither determine which instruction caused the trap nor continue execution of the interrupted instruction stream; it can record the trap that occurred and abort the program.

The meaning of the cause bit field when reading the FCSR is not defined. When a cause bit is written in the FCSR by moving data to it, the corresponding flag bit is also set.

All floating-point operations, whether they cause a trap or not, complete in the sense that they write a result and record exception condition bits in the flag field. When an IEEE exception condition arises during an operation, the default result defined by the IEEE standard is stored (see Table B-4).

A floating-point trap is generated when an exception condition arises during a floating-point operation and the corresponding enable bit is set. A trap will also be generated when a value with corresponding cause and enable bits set is moved into the FCSR. There is no enable for Unimplemented Operation; this exception condition always generates a trap.

The flag bit field is a cumulative report of IEEE exception conditions that arise during instructions that complete. Because all instructions complete in this mode, unlike precise exception mode, the flag bits include exception conditions that cause traps. The flag bits are set to 1 if the corresponding IEEE exception is raised and unchanged otherwise. There is no flag bit for the MIPS Uniplemented Operation exception condition. 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.

B 5.3 Exception Condition Definitions

The five exception conditions defined by the IEEE standard are described in this section. It also describes the MIPS-defined exception condition, Unimplemented Operation, that is used to signal a need for software emulation assistance for 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. 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 (i.e. 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 result is mentioned in each description and summarized inTable B-4.

Table B-4 Default Result for IEEE Exceptions Not Trapped Precisely

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

B 5.3.1 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. The invalid operations are:

- One or both operands is a signaling NaN (except for the non-arithmetic MOV.*fmt* MOVT.*fmt* MOVF.*fmt* MOVN.*fmt* and MOVZ.*fmt* instructions)
- Addition or subtraction: magnitude subtraction of infinities, such as: $(+\infty) + (-\infty)$ or $(-\infty) - (-\infty)$
- Multiplication: $0 \times \infty$, with any signs
- Division: $0 / 0$ or ∞ / ∞, with any signs
- Square root: An operand less than 0 (-0 is a valid operand value).
- Conversion of a floating-point number to a fixed-point format when an overflow, or 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 definition of the compare operation (C.*cond.fmt*) has tables showing the comparisons that do and do not signal the exception.

B 5.3.2 Division By Zero exception

The division by zero exception is signaled on an implemented divide operation 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. The divisions (0/0) and (∞/0) do not cause the division by zero exception. The result of (0/0) is an Invalid Operation exception condition. The result of (∞/0) is a correctly signed infinity.

B 5.3.3 Overflow exception

The overflow exception is signaled when what would have been the magnitude of the rounded floating-point result, were the exponent range unbounded, is larger than the destination format's largest finite number. The result, when no precise trap occurs, is determined by the rounding mode and the sign of the intermediate result as shown in Table B-4.

B 5.3.4 Underflow exception

Two related events contribute to underflow. One is the creation of a tiny non-zero result between $\pm 2^{E\_min}$ which, because it is tiny, may cause some other exception later such as overflow on division. The other is extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers. The IEEE standard permits a choice in how these events are detected, but requires that they must be detected the same way for all operations.

The IEEE standard specifies that "tininess" may be detected either: "after rounding" (when a nonzero result computed as though the exponent range were unbounded would lie strictly between $\pm 2^{E\_min}$), or "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 is detected after rounding.

The IEEE standard specifies that loss of accuracy may be detected as either "denormalization loss" (when the delivered result differs from what would have been computed if the exponent range were unbounded), or "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.

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 (via the FCSR enable field bit), underflow is signaled when tininess is detected regardless of loss of accuracy.

B 5.3.5 Inexact exception

If the rounded result of an operation is not exact or if it overflows without an overflow trap, then the inexact exception is signaled.

B 5.3.6 Unimplemented Operation exception

This MIPS defined (non-IEEE) exception is to provide software emulation support. The architecture is designed to permit a combination of hardware and software to fully implement the architecture. Operations that are not fully supported in hardware cause an Unimplemented Operation exception so that software may perform the operation. There is no enable bit for this condition; it always causes a trap. After the appropriate emulation or other operation is done in a software exception handler, the original instruction stream can be continued.

B 6 Functional Instruction Groups

The FPU instructions are divided into the following functional groups:

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

B 6.1 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:

- FPU general reg ↔ memory (word/doubleword load/store)
- FPU general reg ↔ CPU general reg (word/doubleword move)
- FPU control reg ↔ CPU general reg (word move)

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 will cause an Address Error exception. Regardless of byte-numbering order (endianness), the address of a word or doubleword is the smallest byte address among the bytes 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.

The FPU has loads and stores using the usual register+offset addressing. For the FPU only, there are load and store instructions using register+register addressing.

MIPS I specifies that loads are delayed by one instruction and that proper execution must be insured by observing an instruction scheduling restriction. The instruction immediately following a load into an FPU register *Fn* must not use *Fn* as a source register. The time between the load instruction and the time the data is available is the "load delay slot". If no useful instruction can be put into the load delay slot, then a null operation (NOP) must be inserted.

In MIPS II, this instruction scheduling restriction is removed. Programs will execute correctly when the loaded data is used by the instruction following the load, but this may require extra real cycles. Most processors cannot actually load data quickly enough for immediate use and the processor will be forced to wait until the data is available. Scheduling load delay slots is desirable for performance reasons even when it is not necessary for correctness.

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

Mnemonic	Description	Defined in
LWC1	Load Word to Floating-Point	MIPS I
SWC1	Store Word to Floating-Point	I
LDC1	Load Doubleword to Floating-Point	III
SDC1	Store Doubleword to Floating-Point	III

Table B-6 FPU Loads and Using Register + Register Address Mode

Mnemonic	Description	Defined in
LWXC1	Load Word Indexed to Floating-Point	MIPS IV
SWXC1	Store Word Indexed to Floating-Point	IV
LDXC1	Load Doubleword Indexed to Floating-Point	IV
SDXC1	Store Doubleword Indexed to Floating-Point	IV

Table B-7 FPU Move To/From Instructions

Mnemonic	Description	Defined in
MTC1	Move Word To Floating-Point	MIPS I
MFC1	Move Word From Floating-Point	I
DMTC1	Doubleword Move To Floating-Point	III
DMFC1	Doubleword Move From Floating-Point	III
CTC1	Move Control Word To Floating-Point	I
CFC1	Move Control Word From Floating-Point	I

B 6.2 Arithmetic Instructions

The arithmetic instructions operate on formatted data values. The result of most floating-point arithmetic operations meets the IEEE standard specification for accuracy; a result which is identical to an infinite-precision result 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).

Table B-8 FPU IEEE Arithmetic Operations

Mnemonic	Description	Defined in
ADD.*fmt*	Floating-Point Add	MIPS I
SUB.*fmt*	Floating-Point Subtract	I
MUL.*fmt*	Floating-Point Multiply	I
DIV.*fmt*	Floating-Point Divide	I
ABS.*fmt*	Floating-Point Absolute Value	I
NEG.*fmt*	Floating-Point Negate	I
SQRT.*fmt*	Floating-Point Square Root	II
C.*cond.fmt*	Floating-Point Compare	I

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 by no more than two ulp. Within these error limits, the result of these instructions is implementation specific.

Table B-9 FPU Approximate Arithmetic Operations

Mnemonic	Description	Defined in
RECIP.*fmt*	Floating-Point Reciprocal Approximation	MIPS IV
RSQRT.*fmt*	Floating-Point Reciprocal Square Root Approximation	IV

There are four compound-operation instructions that perform variations of multiply-accumulate: multiply two operands and accumulate to a third operand to produce a result. The accuracy of the result depends which of two alternative arithmetic models is used for the computation. The unrounded model is more accurate than a pair of IEEE operations and the rounded model meets the IEEE specification.

Table B-10 FPU Multiply-Accumulate Arithmetic Operations

Mnemonic	Description	Defined in
MADD.*fmt*	Floating-Point Multiply Add	MIPS IV
MSUB.*fmt*	Floating-Point Multiply Subtract	IV
NMADD.*fmt*	Floating-Point Negative Multiply Add	IV
NMSUB.*fmt*	Floating-Point Negative Multiply Subtract	IV

The initial implementation of the MIPS IV architecture, the R8000 (and future revisions of it), uses the unrounded arithmetic model which does not match the IEEE accuracy specification. All other implementations will use the rounded model which does meet the specification.

- Rounded or non-fused: The product is rounded according to the current rounding mode prior to the accumulation. This model meets the IEEE accuracy specification; the result is numerically identical to the equivalent computation using multiply, add, subtract, and negate instructions.
- Unrounded or fused (R8000 implementation): The product is not rounded and all bits take part in the accumulation. This model does not match the IEEE accuracy requirements; the result is more accurate than the equivalent computation using IEEE multiply, add, subtract, and negate instructions.

B 6.3 Conversion Instructions

There are instructions to perform conversions among the floating-point and fixed-point data types. Each instruction converts values from a number of operand formats to a particular result format. Some convert instructions use the rounding mode specified in the Floating Control and Status Register (FCSR), others specify the rounding mode directly.

Table B-11 FPU Conversion Operations Using the FCSR Rounding Mode

Mnemonic	Description	Defined in
CVT.S.*fmt*	Floating-Point Convert to Single Floating-Point	MIPS I
CVT.D.*fmt*	Floating-Point Convert to Double Floating-Point	I
CVT.W.*fmt*	Floating-Point Convert to Word Fixed-Point	I
CVT.L.*fmt*	Floating-Point Convert to Long Fixed-Point	I

Table B-12 FPU Conversion Operations Using a Directed Rounding Mode

Mnemonic	Description	Defined in
ROUND.W.*fmt*	Floating-Point Round to Word Fixed-Point	II
ROUND.L.*fmt*	Floating-Point Round to Long Fixed-Point	III
TRUNC.W.*fmt*	Floating-Point Truncate to Word Fixed-Point	II
TRUNC.L.*fmt*	Floating-Point Truncate to Long Fixed-Point	III
CEIL.W.*fmt*	Floating-Point Ceiling to Word Fixed-Point	II
CEIL.L.*fmt*	Floating-Point Ceiling to Long Fixed-Point	III
FLOOR.W.*fmt*	Floating-Point Floor to Word Fixed-Point	II
FLOOR.L.*fmt*	Floating-Point Floor to Long Fixed-Point	III

B 6.4 Formatted Operand Value Move Instructions

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

- Unconditional move
- Conditional move that tests an FPU condition code
- Conditional move that tests a CPU general register value against zero

The conditional move instructions operate in a way that may be unexpected. They always force the value in the destination register to become a value of the format specified in the instruction. If the destination register does not contain an operand of the specified format before the conditional move is executed, the contents become undefined. There is more information in Values in FP Registers on page B-13 and in the individual descriptions of the conditional move instructions themselves.

Table B-13 FPU Formatted Operand Move Instructions

Mnemonic	Description	Defined in
MOV.*fmt*	Floating-Point Move	MIPS I

Table B-14 FPU Conditional Move on True/False Instructions

Mnemonic	Description	Defined in
MOVT.*fmt*	Floating-Point Move Conditional on FP True	MIPS IV
MOVF.*fmt*	Floating-Point Move Conditional on FP False	IV

Table B-15 FPU Conditional Move on Zero/Nonzero Instructions

Mnemonic	Description	Defined in
MOVZ.*fmt*	Floating-Point Move Conditional on Zero	MIPS IV
MOVN.*fmt*	Floating-Point Move Conditional on Nonzero	IV

B 6.5 Conditional Branch Instructions

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

All branches have an architectural delay of one instruction. When a branch is taken, the instruction immediately following the branch instruction, in the branch delay slot, is executed before the branch to the target instruction takes place. Conditional branches come in two versions that treat the instruction in the delay slot differently when the branch is not taken and execution falls through. The "branch" instructions execute the instruction in the delay slot, but the "branch likely" instructions do not (they are said to nullify it).

MIPS I defines a single condition code which is implicit in the compare and branch instructions. MIPS IV defines seven additional condition codes and includes the condition code number in the compare and branch instructions. The MIPS IV extension keeps the original condition bit as condition code zero and the extended encoding is compatible with the MIPS I encoding.

Table B-16 FPU Conditional Branch Instructions

Mnemonic	Description	Defined in
BC1T	Branch on FP True	MIPS I
BC1F	Branch on FP False	I
BC1TL	Branch on FP True Likely	II
BC1FL	Branch on FP False Likely	II

B 6.6 Miscellaneous Instructions

B 6.6.1 CPU Conditional Move

There are instructions to move conditionally move one CPU general register to another based on an FPU condition code.

Table B-17 CPU Conditional Move on FPU True/False Instructions

Mnemonic	Description	Defined in
MOVZ	Move Conditional on FP True	MIPS IV
MOVN	Move Conditional on FP False	IV

B 7 Valid Operands for FP Instructions

The floating-point unit arithmetic, conversion, and operand move instructions operate on formatted values with different precision and range limits and produce formatted values for results. Each representable value in each format has a binary encoding that is read from or stored to memory. The *fmt* or *fmt3* field of the instruction encodes the operand format required for the instruction. A conversion instruction specifies the result type in the *function* field; the result of other operations is the same format as the operands. The encoding of the *fmt* and *fmt3* fields is shown in Table B-18.

Table B-18 FPU Operand Format Field (fmt, fmt3) Decoding

fmt	*fmt3*	Instruction Mnemonic	Size name	Size bits	data type
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
22–31	6-7	Reserved			

Each type of arithmetic or conversion instruction is valid for operands of selected formats. A summary of the computational and operand move instructions and the formats valid for each of them is listed in Table B-19. Implementations must support combinations that are valid either directly in hardware or through emulation in an exception handler.

The result of an instruction using operand formats marked "U" is not currently specified by this architecture and will cause an exception. They are available for future extensions to the architecture. The exact exception mechanism used is processor specific. Most implementations report this as an Unimplemented Operation for a Floating Point exception. Other implementations report these combinations as Reserved Instruction exceptions.

The result of an instruction using operand formats marked "i" are invalid and an attempt to execute such an instruction has an undefined result.

Table B-19 Valid Formats for FPU Operations

Mnemonic	Operation	operand fmt float S	float D	fixed W	fixed L	COP1 function value	COP1 X op4 value
ABS	Absolute value	•	•	U	U	5	
ADD	Add	•	•	U	U	0	
C.*cond*	Floating-point compare	•	•	U	U	48–63	
CEIL.L, (CEIL.W)	Convert to longword fixed-point, round toward +∞	•	•	i	i	10 (14)	
CVT.D	Convert to double floating-point	•	i	•	•	33	
CVT.L	Convert to longword fixed-point	•	•	i	i	37	
CVT.S	Convert to single floating-point	i	•	•	•	32	
CVT.W	Convert to 32-bit fixed-point	•	•	i	i	36	
DIV	Divide	•	•	U	U	3	
FLOOR.L, (FLOOR.W)	Convert to longword fixed-point, round toward -∞	•	•	i	i	11 (15)	
MADD	Multiply-Add	•	•	U	U		4
MOV	Move Register	•	•	i	i	6	

Mnemonic	Operation	operand fmt float S	D	fixed W	L	COP1 func-tion value	COP1 X op4 value
MOVC	FP Move Conditional on condition	•	•	i	i	17	
MOVN	FP Move Conditional on GPR ≠ zero	•	•	i	i	19	
MOVZ	FP Move Conditional on GPR = zero	•	•	i	i	18	
MSUB	Multiply-Subtract	•	•	U	U		5
MUL	Multiply	•	•	U	U	2	
NEG	Negate	•	•	U	U	7	
NMADD	Negative multiply-Add	•	•	U	U		6
NMSUB	Negative multiply-Subtract	•	•	U	U		7
RECIP	Reciprocal approximation	•	•	U	U	21	
ROUND.L, (ROUND.W)	Convert to longword fixed-point, round to nearest/even	•	•	i	i	8 (12)	
RSQRT	Reciprocal square root approximation	•	•	U	U	22	
SQRT	Square root	•	•	U	U	4	
SUB	Subtract	•	•	U	U	1	
TRUNC.L (TRUNC.W)	Convert to longword fixed-point, round toward zero	•	•	i	i	9 (13)	
Key:	• – Valid. U – Unimplemented or Reserved. i – Invalid.						

B 8 Description of an Instruction

For the FPU instruction detail documentation, all variable subfields in an instruction format (such as *fs*, *ft*, *immediate*, and so on) are shown in lower-case. The instruction name (such as ADD, SUB, and so on) is shown in upper-case.

For the sake of clarity, we sometimes use an alias for a variable subfield in the formats of specific instructions. For example, we use *rs* = *base* in the format for load and store instructions. Such an alias is always lower case, since it refers to a variable subfield.

In some instructions, the instruction subfields *op* and *function* can have constant 6-bit values. When reference is made to these instructions, upper-case mnemonics are used. For instance, in the floating-point ADD instruction we use *op* = COP1 and *function* = ADD. In other cases, a single field has both fixed and variable subfields, so the name contains both upper and lower case characters. Bit encodings for mnemonics are shown at the end of this section, and are also included with each individual instruction.

B 9 Operation Notation Conventions and Functions

The instruction description includes an *Operation* section that describes the operation of the instruction in a pseudocode. The pseudocode and terms used in the description are described in Operation Section Notation and Functions on page A-18.

B 10 Individual FPU Instruction Descriptions

The FP instructions are described in alphabetic order. See Description of an Instruction on page A-15 for a description of the information in each instruction description.

ABS.fmt

Floating-Point Absolute Value

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	ABS 0 0 0 1 0 1
6	5	5	5	5	6

Format: ABS.S fd, fs
ABS.D fd, fs

MIPS I

Purpose: To compute the absolute value of an FP value.

Description: fd ← absolute(fs)

The absolute value of the value in FPR *fs* is placed in FPR *fd*. The operand and result are values in format *fmt*.

This operation is arithmetic; a NaN operand signals invalid operation.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
StoreFPR(fd, fmt, AbsoluteValue(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Unimplemented Operation
  Invalid Operation

Floating-Point Add ADD.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	ft	fs	fd	ADD 0 0 0 0 0 0
6	5	5	5	5	6

Format: ADD.S fd, fs, ft
ADD.D fd, fs, ft

MIPS I

Purpose: To add FP values.

Description: fd ← fs + ft

The value in FPR *ft* is added to the value in FPR *fs*. The result is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR *fd*. The operands and result are values in format *fmt*.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format *fmt*; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

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

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Unimplemented Operation
- Invalid Operation
- Inexact
- Overflow
- Underflow

BC1F

Branch on FP False

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

Format: BC1F offset (cc = 0 implied) MIPS I

BC1F cc, offset MIPS IV

Purpose: To test an FP condition code and do a PC-relative conditional branch.

Description: if (cc = 0) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the FP condition code bit *cc* is false (0), branch to the effective target address after the instruction in the delay slot is executed

An FP condition code is set by the FP compare instruction, C.*cond.fmt*.

The MIPS I architecture defines a single floating-point condition code, implemented as the coprocessor 1 condition signal (Cp1Cond) and the C bit in the FP *Control and Status* register. MIPS I, II, and III architectures must have the *cc* field set to 0, which is implied by the first format in the *Format* section.

The MIPS IV architecture adds seven more condition code bits to the original condition code 0. FP compare and conditional branch instructions specify the condition code bit to set or test. Both assembler formats are valid for MIPS IV.

Restrictions:

MIPS I, II, III: There must be at least one instruction between the compare instruction that sets a condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.

MIPS IV: None.

Branch on FP False **BC1F**

Operation:

MIPS I, II, and III define a single condition code; MIPS IV adds 7 more condition codes.This operation specification is for the general "Branch On Condition" operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

MIPS I

```
I-1:    condition ← COC[1] = tf
I:      target_offset← (offset_15)^(GPRLEN-(16+2)) || offset || 0^2
I+1:    if condition then
            PC ← PC + target
        endif
```

MIPS II and MIPS III:

```
I-1:    condition ← COC[1] = tf
I:      target_offset← (offset_15)^(GPRLEN-(16+2)) || offset || 0^2
I+1:    if condition then
            PC ← PC + target
        else if nd then
            NullifyCurrentInstruction()
        endif
```

MIPS IV:

```
I:      condition ← FCC[cc] = tf
        target_offset← (offset_15)^(GPRLEN-(16+2)) || offset || 0^2
I+1:    if condition then
            PC ← PC + target
        else if nd then
            NullifyCurrentInstruction()
        endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Unimplemented Operation

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BC1FL Branch on FP False Likely

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

Format: BC1FL offset (cc = 0 implied) MIPS II
BC1FL cc, offset MIPS IV

Purpose: To test an FP condition code and do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (cc = 0) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the FP condition code bit *cc* is false (0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

An FP condition code is set by the FP compare instruction, C.*cond.fmt*.

The MIPS I architecture defines a single floating-point condition code, implemented as the coprocessor 1 condition signal (Cp1Cond) and the C bit in the FP *Control and Status* register. MIPS I, II, and III architectures must have the *cc* field set to 0, which is implied by the first format in the *Format* section.

The MIPS IV architecture adds seven more condition code bits to the original condition code 0. FP compare and conditional branch instructions specify the condition code bit to set or test. Both assembler formats are valid for MIPS IV.

Restrictions:

MIPS II, III: There must be at least one instruction between the compare instruction that sets a condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.

MIPS IV: None.

Operation:

MIPS II, and III define a single condition code; MIPS IV adds 7 more condition codes.This operation specification is for the general "Branch On Condition" operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

MIPS II and MIPS III:

```
I-1:   condition ← COC[1] = tf
I:     target_offset← (offset_15)^(GPRLEN-(16+2)) || offset || 0^2
I+1:   if condition then
           PC ← PC + target
       else if nd then
           NullifyCurrentInstruction()
       endif
```

MIPS IV:

```
I:     condition ← FCC[cc] = tf
       target_offset← (offset_15)^(GPRLEN-(16+2)) || offset || 0^2
I+1:   if condition then
           PC ← PC + target
       else if nd then
           NullifyCurrentInstruction()
       endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Unimplemented Operation

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

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

Format:	BC1T offset	(cc = 0 implied)	MIPS I
	BC1T cc, offset		MIPS IV

Purpose: To test an FP condition code and do a PC-relative conditional branch.

Description: if (cc = 1) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the FP condition code bit *cc* is true (1), branch to the effective target address after the instruction in the delay slot is executed

An FP condition code is set by the FP compare instruction, C.*cond.fmt*.

The MIPS I architecture defines a single floating-point condition code, implemented as the coprocessor 1 condition signal (Cp1Cond) and the C bit in the FP *Control and Status* register. MIPS I, II, and III architectures must have the *cc* field set to 0, which is implied by the first format in the *Format* section.

The MIPS IV architecture adds seven more condition code bits to the original condition code 0. FP compare and conditional branch instructions specify the condition code bit to set or test. Both assembler formats are valid for MIPS IV.

Restrictions:

MIPS I, II, III: There must be at least one instruction between the compare instruction that sets a condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.

MIPS IV: None

Operation:

MIPS I, II, and III define a single condition code; MIPS IV adds 7 more condition codes.This operation specification is for the general “Branch On Condition” operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

MIPS I

```
I-1:   condition ← COC[1] = tf
I:     target ← (offset15)^(GPRLEN-(16+2)) || offset || 0^2
I+1:   if condition then
           PC ← PC + target
       endif
```

MIPS II and MIPS III:

```
I-1:   condition ← COC[1] = tf
I:     target ← (offset15)^(GPRLEN-(16+2)) || offset || 0^2
I+1:   if condition then
           PC ← PC + target
       else if nd then
           NullifyCurrentInstruction()
       endif
```

MIPS IV:

```
I:     condition ← FCC[cc] = tf
       target ← (offset15)^(GPRLEN-(16+2)) || offset || 0^2
I+1:   if condition then
           PC ← PC + target
       else if nd then
           NullifyCurrentInstruction()
       endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Unimplemented Operation

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BC1TL

Branch on FP True Likely

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

Format: BC1TL offset (cc = 0 implied) MIPS II
BC1TL cc, offset MIPS IV

Purpose: To test an FP condition code and do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (cc = 1) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the FP condition code bit *cc* is true (1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

An FP condition code is set by the FP compare instruction, C.*cond.fmt*.

The MIPS I architecture defines a single floating-point condition code, implemented as the coprocessor 1 condition signal (Cp1Cond) and the C bit in the FP *Control and Status* register. MIPS I, II, and III architectures must have the *cc* field set to 0, which is implied by the first format in the *Format* section.

The MIPS IV architecture adds seven more condition code bits to the original condition code 0. FP compare and conditional branch instructions specify the condition code bit to set or test. Both assembler formats are valid for MIPS IV.

Restrictions:

MIPS II, III: There must be at least one instruction between the compare instruction that sets a condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.

MIPS IV: None.

Branch on FP True Likely BC1TL

Operation:

MIPS II, and III define a single condition code; MIPS IV adds 7 more condition codes.This operation specification is for the general "Branch On Condition" operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

MIPS II and MIPS III:

I-1: condition ← COC[1] = tf
I: target ← $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || 0^2
I+1 :if condition then
PC ← PC + target
else if nd then
NullifyCurrentInstruction()
endif

MIPS IV:

I: condition ← FCC[cc] = tf
target ← $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || 0^2
I+1 :if condition then
PC ← PC + target
else if nd then
NullifyCurrentInstruction()
endif

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
Unimplemented Operation

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

C.cond.fmt

Floating-Point Compare

31–26	25–21	20–16	15–11	10–8	7–6	5–4	3–0
COP1 010001	fmt	ft	fs	cc	0 00	FC 11	cond
6	5	5	5	3	2	2	4

Format: C.cond.S fs, ft (cc = 0 implied) **MIPS I**
C.cond.D fs, ft (cc = 0 implied)
C.cond.S cc, fs, ft **MIPS IV**
C.cond.D cc, fs, ft

Purpose: To compare FP values and record the Boolean result in a condition code.

Description: cc ← fs *compare_cond* ft

The value in FPR *fs* is compared to the value in FPR *ft*; the values are in format *fmt*. The comparison is exact and neither overflows nor underflows. If the comparison specified by $cond_{2..1}$ is true for the operand values, then the result is true, otherwise it is false. If no exception is taken, the result is written into condition code *cc*; true is 1 and false is 0.

If $cond_3$ is set and at least one of the values is a NaN, an Invalid Operation condition is raised; the result depends on the FP exception model currently active.

- Precise exception model: The Invalid Operation flag is set in the FCSR. If the Invalid Operation enable bit is set in the FCSR, no result is written and an Invalid Operation exception is taken immediately. Otherwise, the Boolean result is written into condition code *cc*.
- Imprecise exception model (R8000 normal mode): The Boolean result is written into condition code *cc*. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

There are four mutually exclusive ordering relations for comparing floating-point values; one relation is always true and the others are false. The familiar relations are *greater than*, *less than*, and *equal*. In addition, the IEEE floating-point standard defines the relation *unordered* which is true when at least one operand value is NaN; NaN compares unordered with everything, including itself. Comparisons ignore the sign of zero, so +0 equals -0.

The comparison condition is a logical predicate, or equation, of the ordering relations such as “less than or equal”, “equal”, “not less than”, or “unordered or equal”. Compare distinguishes sixteen comparison predicates. The Boolean result of the instruction is obtained by substituting the Boolean value of each ordering relation for the two FP values into equation. If the *equal* relation is true, for example, then all four example predicates above would yield a true result. If the *unordered* relation is true then only the final predicate, “unordered or equal” would yield a true result.

Logical negation of a compare result allows eight distinct comparisons to test for sixteen predicates as shown in Table B-20. Each mnemonic tests for both a predicate and its logical negation. For each mnemonic, compare tests the truth of the first predicate. When the first predicate is true, the result is true as shown in the "if predicate is true" column (note that the False predicate is never true and False/True do not follow the normal pattern). When the first predicate is true, the second predicate must be false, and vice versa. The truth of the second predicate is the logical negation of the instruction result. After a compare instruction, test for the truth of the first predicate with the Branch on FP True (BC1T) instruction and the truth of the second with Branch on FP False (BC1F).

Table B-20 FPU Comparisons Without Special Operand Exceptions

Instr	Comparison Predicate					Comparison CC Result		Instr	
cond Mne-monic	name of predicate and logically negated predicate (abbreviation)	relation values				If pred-icate is true	Inv Op excp if Q NaN	cond field	
		>	<	=	?			3	2..0
F	False [this predicate is always False,	F	F	F	F	F	No	0	0
	True (T) it never has a True result]	T	T	T	T				
UN	Unordered	F	F	F	T	T			1
	Ordered (OR)	T	T	T	F	F			
EQ	Equal	F	F	T	F	T			2
	Not Equal (NEQ)	T	T	F	T	F			
UEQ	Unordered or Equal	F	F	T	T	T			3
	Ordered or Greater than or Less than (OGL)	T	T	F	F	F			
OLT	Ordered or Less Than	F	T	F	F	T			4
	Unordered or Greater than or Equal (UGE)	T	F	T	T	F			
ULT	Unordered or Less Than	F	T	F	T	T			5
	Ordered or Greater than or Equal (OGE)	T	F	T	F	F			
OLE	Ordered or Less than or Equal	F	T	T	F	T			6
	Unordered or Greater Than (UGT)	T	F	F	T	F			
ULE	Unordered or Less than or Equal	F	T	T	T	T			7
	Ordered or Greater Than (OGT)	T	F	F	F	F			

key: "?" = *unordered*, ">" = *greater than*, "<" = *less than*, "=" is *equal*, "T" = True, "F" = False

There is another set of eight compare operations, distinguished by a $cond_3$ value of 1, testing the same sixteen conditions. For these additional comparisons, if at least one of the operands is a NaN, including Quiet NaN, then an Invalid Operation condition is raised. If the Invalid Operation condition is enabled in the FCSR, then an Invalid Operation exception occurs.

Table B-21 FPU Comparisons With Special Operand Exceptions for QNaNs

Instr	Comparison Predicate					Comparison CC Result		Instr	
cond Mne-monic	name of predicate and logically negated predicate (abbreviation)	relation values				If pred-icate is true	Inv Op excp if Q NaN	cond field	
		>	<	=	?			3	2..0
SF	Signaling False [this predicate always False]	F	F	F	F	F	Yes	1	0
	Signaling True (ST)	T	T	T	T				
NGLE	Not Greater than or Less than or Equal	F	F	F	T	T			1
	Greater than or Less than or Equal (GLE)	T	T	T	F	F			
SEQ	Signaling Equal	F	F	T	F	T			2
	Signaling Not Equal (SNE)	T	T	F	T	F			
NGL	Not Greater than or Less than	F	F	T	T	T			3
	Greater than or Less than (GL)	T	T	F	F	F			
LT	Less than	F	T	F	F	T			4
	Not Less Than (NLT)	T	F	T	T	F			
NGE	Not Greater than or Equal	F	T	F	T	T			5
	Greater than or Equal (GE)	T	F	T	F	F			
LE	Less than or Equal	F	T	T	F	T			6
	Not Less than or Equal (NLE)	T	F	F	T	F			
NGT	Not Greater than	F	T	T	T	T			7
	Greater than (GT)	T	F	F	F	F			

key: "?" = *unordered*, ">" = *greater than*, "<" = *less than*, "=" is *equal*, "T" = True, "F" = False

The instruction encoding is an extension made in the MIPS IV architecture. In previous architecture levels the *cc* field for this instruction must be 0.

The MIPS I architecture defines a single floating-point condition code, implemented as the coprocessor 1 condition signal (Cp1Cond) and the C bit in the FP *Control and Status* register. MIPS I, II, and III architectures must have the *cc* field set to 0, which is implied by the first format in the *Format* section.

The MIPS IV architecture adds seven more condition code bits to the original condition code 0. FP compare and conditional branch instructions specify the condition code bit to set or test. Both assembler formats are valid for MIPS IV.

Restrictions:

The fields *fs* and *ft* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format *fmt*; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

MIPS I, II, III: There must be at least one instruction between the compare instruction that sets a condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.

Operation:

```
if NaN(Value FPR(fs, fmt)) or NaN(ValueFPR(ft, fmt)) then
    less ← false
    equal ← false
    unordered ← true
    if t then
        SignalException(InvalidOperation)
    endif
else
    less ← ValueFPR(fs, fmt) < ValueFPR(ft, fmt)
    equal ← ValueFPR(fs, fmt) = ValueFPR(ft, fmt)
    unordered ← false
endif
condition ← (cond2 and less) or (cond1 and equal) or (cond0 and unordered)
FCC[cc] ← condition
if cc = 0 then
    COC[1] ← condition
endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Unimplemented Operation
    Invalid Operation

Programming Notes:

FP computational instructions, including compare, that receive an operand value of Signaling NaN, will raise the Invalid Operation condition. The comparisons that raise the Invalid Operation condition for Quiet NaNs in addition to SNaNs, permit a simpler programming model if NaNs are errors. Using these compares, programs do not need explicit code to check for QNaNs causing the *unordered* relation. Instead, they take an exception and allow the exception handling system to deal with the error when it occurs. For example, consider a comparison in which we want to know if two numbers are equal, but for which unordered would be an error.

```
# comparisons using explicit tests for QNaN
    c.eq.d  $f2,$f4  # check for equal
    nop
    bc1t    L2       # it is equal
    c.un.d  $f2,$f4  # it is not equal, but might be unordered
    bc1t    ERROR# unordered goes off to an error handler
# not-equal-case code here
    ...
# equal-case code here
L2:
# ----------------------------------------------------------------
# comparison using comparisons that signal QNaN
    c.seq.d $f2,$f4  # check for equal
    nop
    bc1t    L2       # it is equal
    nop
# it is not unordered here...
# not-equal-case code here
    ...
#equal-case code here
L2:
```

Floating-Point Ceiling Convert to Long Fixed-Point CEIL.L.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CEIL.L 0 0 1 0 1 0
6	5	5	5	5	6

Format: CEIL.L.S fd, fs   MIPS III
CEIL.L.D fd, fs

Purpose: To convert an FP value to 64-bit fixed-point, rounding up.

Description: fd ← convert_and_round(fs)

The value in FPR *fs* in format *fmt*, is converted to a value in 64-bit long fixed-point format rounding toward +∞ (rounding mode 2). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: The Invalid Operation flag is set in the FCSR. If the Invalid Operation enable bit is set in the FCSR, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}–1$, is written to *fd*.
- Imprecise exception model (R8000 normal mode): The default result, $2^{63}–1$, is written to *fd*. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Invalid Operation   Unimplemented Operation
    Inexact   Overflow

CEIL.W.fmt **Floating-Point Ceiling Convert to Word Fixed-Point**

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CEIL.W 0 0 1 1 1 0
6	5	5	5	5	6

Format: CEIL.W.S fd, fs **MIPS II**
CEIL.W.D fd, fs

Purpose: To convert an FP value to 32-bit fixed-point, rounding up.

Description: fd ← convert_and_round(fs)

The value in FPR *fs* in format *fmt*, is converted to a value in 32-bit word fixed-point format rounding toward +∞ (rounding mode 2). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: The Invalid Operation flag is set in the FCSR. If the Invalid Operation enable bit is set in the FCSR, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}–1$, is written to *fd*.
- Imprecise exception model (R8000 normal mode): The default result, $2^{31}–1$, is written to *fd*. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Invalid Operation
    Unimplemented Operation
    Inexact
    Overflow

Move Control Word from Floating-Point CFC1

31–26	25–21	20–16	15–11	10–0
COP1 0 1 0 0 0 1	CF 0 0 0 1 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format: CFC1 rt, fs **MIPS I**

Purpose: To copy a word from an FPU control register to a GPR.

Description: rt ← FP_Control[fs]

Copy the 32-bit word from FP (coprocessor 1) control register *fs* into GPR *rt*, sign-extending it if the GPR is 64 bits.

Restrictions:

There are only a couple control registers defined for the floating-point unit. The result is not defined if *fs* specifies a register that does not exist.

For MIPS I, MIPS II, and MIPS III, the contents of GPR *rt* are undefined for the instruction immediately following CFC1.

Operation: MIPS I - III

```
I:      temp    ← FCR[fs]
I+1:    GPR[rt] ← sign_extend(temp)
```

Operation: MIPS IV

```
        temp    ← FCR[fs]
        GPR[rt] ← sign_extend(temp)
```

Exceptions:

Coprocessor Unusable

31–26	25–21	20–16	15–11	10–0
COP1 0 1 0 0 0 1	CT 0 0 1 1 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format: CTC1 rt, fs **MIPS I**

Purpose: To copy a word from a GPR to an FPU control register.

Description: FP_Control[fs] ← rt

Copy the low word from GPR *rt* into FP (coprocessor 1) control register *fs*.

Writing to control register 31, the *Floating-Point Control and Status Register* or FCSR, causes the appropriate exception if any cause bit and its corresponding enable bit are both set. The register will be written before the exception occurs.

Restrictions:

There are only a couple control registers defined for the floating-point unit. The result is not defined if *fs* specifies a register that does not exist.

For MIPS I, MIPS II, and MIPS III, the contents of floating-point control register *fs* are undefined for the instruction immediately following CTC1.

Operation: MIPS I - III

I: temp ← $GPR[rt]_{31..0}$
I+ 1 :FCR[fs] ← temp
COC[1] ← $FCR[31]_{23}$

Operation: MIPS IV

temp ← $GPR[rt]_{31..0}$
FCR[fs] ← temp
COC[1] ← $FCR[31]_{23}$

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Unimplemented Operation
- Invalid Operation
- Division-by-zero
- Inexact
- Overflow
- Underflow

Floating-Point Convert to Double Floating-Point **CVT.D.fmt**

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CVT.D 1 0 0 0 0 1
6	5	5	5	5	6

Format: CVT.D.S fd, fs — MIPS I
CVT.D.W fd, fs
CVT.D.L fd, fs — MIPS III

Purpose: To convert an FP or fixed-point value to double FP.

Description: fd ← convert_and_round(fs)

The value in FPR *fs* in format *fmt* is converted to a value in double floating-point format rounded according to the current rounding mode in FCSR. The result is placed in FPR *fd*.

If *fmt* is S or W, then the operation is always exact.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for double floating-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

StoreFPR (fd, D, ConvertFmt(ValueFPR(fs, fmt), fmt, D))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow
- Underflow

CVT.L.fmt — Floating-Point Convert to Long Fixed-Point

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CVT.L 1 0 0 1 0 1
6	5	5	5	5	6

Format: CVT.L.S fd, fs **MIPS III**
CVT.L.D fd, fs

Purpose: To convert an FP value to a 64-bit fixed-point.

Description: fd ← convert_and_round(fs)

Convert the value in format *fmt* in FPR *fs* to long fixed-point format, round according to the current rounding mode in FCSR, and place the result in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active:

- Precise exception model: The Invalid Operation flag is set in the FCSR. If the Invalid Operation enable bit is set in the FCSR, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}–1$, is written to *fd*.
- Imprecise exception model (R8000 normal mode): The default result, $2^{63}–1$, is written to *fd*. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

StoreFPR (fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Invalid Operation
  Unimplemented Operation
  Inexact
  Overflow

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CVT.S 1 0 0 0 0 0
6	5	5	5	5	6

Format: CVT.S.D fd, fs — MIPS I
CVT.S.W fd, fs
CVT.S.L fd, fs — MIPS III

Purpose: To convert an FP or fixed-point value to single FP.

Description: fd ← convert_and_round(fs)

The value in FPR *fs* in format *fmt* is converted to a value in single floating-point format rounded according to the current rounding mode in FCSR. The result is placed in FPR *fd*.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for single floating-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

StoreFPR(fd, S, ConvertFmt(ValueFPR(fs, fmt), fmt, S))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow
- Underflow

CVT.W.fmt

Floating-Point Convert to Word Fixed-Point

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CVT.W 1 0 0 1 0 0
6	5	5	5	5	6

Format: CVT.W.S fd, fs **MIPS I**
CVT.W.D fd, fs

Purpose: To convert an FP value to 32-bit fixed-point.

Description: fd ← convert_and_round(fs)

The value in FPR *fs* in format *fmt* is converted to a value in 32-bit word fixed-point format rounded according to the current rounding mode in FCSR. The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: The Invalid Operation flag is set in the FCSR. If the Invalid Operation enable bit is set in the FCSR, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}–1$, is written to *fd*.
- Imprecise exception model (R8000 normal mode): The default result, $2^{31}–1$, is written to *fd*. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Invalid Operation
  Unimplemented Operation
  Inexact
  Overflow

Floating-Point Divide DIV.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	ft	fs	fd	DIV 0 0 0 0 1 1
6	5	5	5	5	6

Format: DIV.S fd, fs, ft — MIPS I
DIV.D fd, fs, ft

Purpose: To divide FP values.

Description: fd ← fs / ft

The value in FPR *fs* is divided by the value in FPR *ft*. The result is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR *fd*. The operands and result are values in format *fmt*.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format *fmt*; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

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

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point

Inexact	Unimplemented Operation
Division-by-zero	Invalid Operation
Overflow	Underflow

31–26	25–21	20–16	15–11	10–0
COP1 0 1 0 0 0 1	DMF 0 0 0 0 1	rt	fs	0 0 0 0 0 0 0 0 0 0 00
6	5	5	5	11

Format: DMFC1 rt, fs **MIPS III**

Purpose: To copy a doubleword from an FPR to a GPR.

Description: rt ← fs

The doubleword contents of FPR *fs* are placed into GPR *rt*.

If the coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR *fs* is held in an even/odd register pair. The low word is taken from the even register *fs* and the high word is from *fs*+1.

Restrictions:

If *fs* does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Registers on page B-6.

For MIPS III, the contents of GPR *rt* are undefined for the instruction immediately following DMFC1.

Operation: MIPS I - III

```
I:    if SizeFGR() = 64 then          /* 64-bit wide FGRs */
          data ← FGR[fs]
      elseif fs0 = 0 then             /* valid specifier, 32-bit wide FGRs */
          data ← FGR[fs+1] || FGR[fs]
      else                            /* undefined for odd 32-bit FGRs */
          UndefinedResult()
      endif
I+1:  GPR[rt] ← data
```

Operation: MIPS IV

```
      if SizeFGR() = 64 then          /* 64-bit wide FGRs */
          data ← FGR[fs]
      elseif fs0 = 0 then             /* valid specifier, 32-bit wide FGRs */
          data ← FGR[fs+1] || FGR[fs]
      else                            /* undefined for odd 32-bit FGRs */
          UndefinedResult()
      endif
      GPR[rt] ← data
```

Exceptions:

Reserved Instruction
Coprocessor Unusable

Doubleword Move To Floating-Point **DMTC1**

31–26	25–21	20–16	15–11	10–0
COP1 0 1 0 0 0 1	DMT 0 0 1 0 1	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format: DMTC1 rt, fs **MIPS III**

Purpose: To copy a doubleword from a GPR to an FPR.

Description: fs ← rt

The doubleword contents of GPR *rt* are placed into FPR *fs*.

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR *fs* is held in an even/odd register pair. The low word is placed in the even register *fs* and the high word is placed in *fs*+1.

Restrictions:

If *fs* does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Registers on page B-6.

For MIPS III, the contents of FPR *fs* are undefined for the instruction immediately following DMTC1.

Operation: MIPS I - III

```
I:    data ← GPR[rt]
I+1:  if SizeFGR() = 64 then          /* 64-bit wide FGRs */
          FGR[fs] ← data
      elseif fs_0 = 0 then            /* valid specifier, 32-bit wide FGRs */
          FGR[fs+1] ← data_63..32
          FGR[fs] ← data_31..0
      else                            /* undefined result for odd 32-bit FGRs */
          UndefinedResult()
      endif
```

Operation: MIPS IV

```
      data ← GPR[rt]
      if SizeFGR() = 64 then          /* 64-bit wide FGRs */
          FGR[fs] ← data
      elseif fs_0 = 0 then            /* valid specifier, 32-bit wide FGRs */
          FGR[fs+1] ← data_63..32
          FGR[fs] ← data_31..0
      else                            /* undefined result for odd 32-bit FGRs */
          UndefinedResult()
      endif
```

Exceptions:

Reserved Instruction
Coprocessor Unusable

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	FLOOR.L 0 0 1 0 1 1
6	5	5	5	5	6

Format: FLOOR.L.S fd, fs **MIPS III**
FLOOR.L.D fd, fs

Purpose: To convert an FP value to 64-bit fixed-point, rounding down.

Description: fd ← convert_and_round(fs)

The value in FPR *fs* in format *fmt*, is converted to a value in 64-bit long fixed-point format rounding toward -∞ (rounding mode 3). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: The Invalid Operation flag is set in the FCSR. If the Invalid Operation enable bit is set in the FCSR, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}–1$, is written to *fd*.
- Imprecise exception model (R8000 normal mode): The default result, $2^{63}–1$, is written to *fd*. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow

FLOOR.W.fmt **Floating-Point Floor Convert to Word Fixed-Point**

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	FLOOR.W 0 0 1 1 1 1
6	5	5	5	5	6

Format: FLOOR.W.S fd, fs **MIPS II**
FLOOR.W.D fd, fs

Purpose: To convert an FP value to 32-bit fixed-point, rounding down.

Description: fd ← convert_and_round(fs)

The value in FPR *fs* in format *fmt*, is converted to a value in 32-bit word fixed-point format rounding toward $-\infty$ (rounding mode 3). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: The Invalid Operation flag is set in the FCSR. If the Invalid Operation enable bit is set in the FCSR, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *fd*.
- Imprecise exception model (R8000 normal mode): The default result, $2^{31}-1$, is written to *fd*. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Invalid Operation   Unimplemented Operation
  Inexact   Overflow

Load Doubleword to Floating-Point LDC1

31–26	25–21	20–16	15–0
LDC1 1 1 0 1 0 1	base	ft	offset
6	5	5	16

Format: LDC1 ft, offset(base) **MIPS II**

Purpose: To load a doubleword from memory to an FPR.

Description: ft ← memory[base+offset]

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in FPR *ft*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR *ft* is held in an even/odd register pair. The low word is placed in the even register *ft* and the high word is placed in *ft*+1.

Restrictions:

If *ft* does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Registers on page B-6.

An Address Error exception occurs if $EffectiveAddress_{2..0} \neq 0$ (not doubleword-aligned).

MIPS IV: The low-order 3 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
data ← LoadMemory(uncached, DOUBLEWORD, pAddr, vAddr, DATA)
if SizeFGR() = 64 then                /* 64-bit wide FGRs */
    FGR[ft] ← data
elseif ft0 = 0 then                   /* valid specifier, 32-bit wide FGRs */
    FGR[ft+1] ← data63..32
    FGR[ft] ← data31..0
else                                  /* undefined result for odd 32-bit FGRs */
    UndefinedResult()
endif
```

Exceptions:

Coprocessor unusable
Reserved Instruction
TLB Refill, TLB Invalid
Address Error

LDXC1 **Load Doubleword Indexed to Floating-Point**

31–26	25–21	20–16	15–11	10–6	5–0
COP1X 0 1 0 0 1 1	base	index	0	fd	LDXC1 0 0 0 0 0 1
6	5	5	5	5	6

Format: LDXC1 fd, index(base) **MIPS IV**

Purpose: To load a doubleword from memory to an FPR (GPR+GPR addressing).

Description: fd ← memory[base+index]

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in FPR *fd*. The contents of GPR *index* and GPR *base* are added to form the effective address.

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR *fd* is held in an even/odd register pair. The low word is placed in the even register *fd* and the high word is placed in *fd*+1.

Restrictions:

If *fd* does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Registers on page B-6.

The Region bits of the effective address must be supplied by the contents of *base*. If $\text{EffectiveAddress}_{63..62} \neq base_{63..62}$, the result is undefined.

An Address Error exception occurs if $\text{EffectiveAddress}_{2..0} \neq 0$ (not doubleword-aligned).

MIPS IV: The low-order 3 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation:

```
vAddr ← GPR[base] + GPR[index]
if vAddr2..0 ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
mem ← LoadMemory(unchched, DOUBLEWORD, pAddr, vAddr, DATA)
if SizeFGR() = 64 then                /* 64-bit wide FGRs */
    FGR[fd] ← data
elseif fd0 = 0 then                   /* valid specifier, 32-bit wide FGRs */
    FGR[fd+1] ← data63..32
    FGR[fd] ← data31..0
else                                  /* undefined result for odd 32-bit FGRs */
    UndefinedResult()
endif
```

Exceptions:

TLB Refill, TLB Invalid
Address Error
Reserved Instruction
Coprocessor Unusable

LWC1 — Load Word to Floating-Point

31–26	25–21	20–16	15–0
LWC1 110001	base	ft	offset
6	5	5	16

Format: LWC1 ft, offset(base) MIPS I

Purpose: To load a word from memory to an FPR.

Description: ft ← memory[base+offset]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of coprocessor 1 general register *ft*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

If coprocessor 1 general registers are 64-bits wide, bits 63..32 of register *ft* become undefined. See Floating-Point Registers on page B-6.

Restrictions:

An Address Error exception occurs if $\text{EffectiveAddress}_{1..0} \neq 0$ (not word-aligned).

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32-bit Processors

```
I:    /* "mem" is aligned 64-bits from memory. Pick out correct bytes. */
      vAddr ← sign_extend(offset) + GPR[base]
      if vAddr1..0 ≠ 0^2 then SignalException(AddressError) endif
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
      mem ← LoadMemory(uncached, WORD, pAddr, vAddr, DATA)
I+1:  FGR[ft] ← mem
```

Operation: 64-bit Processors

```
/* "mem" is aligned 64-bits from memory. Pick out correct bytes. */
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
mem ← LoadMemory(uncached, WORD, pAddr, vAddr, DATA)
bytesel ← vAddr2..0 xor (BigEndianCPU || 0^2)
if SizeFGR() = 64 then                                  /* 64-bit wide FGRs */
    FGR[ft] ← undefined^32 || mem31+8*bytesel..8*bytesel
else                                                    /* 32-bit wide FGRs */
    FGR[ft] ← mem31+8*bytesel..8*bytesel
endif
```

Exceptions:

Coprocessor unusable
Reserved Instruction
TLB Refill, TLB Invalid
Address Error

31–26	25–21	20–16	15–11	10–6	5–0
COP1X 0 1 0 0 1 1	base	index	0	fd	LWXC1 0 0 0 0 0 0
6	5	5	5	5	6

Format: LWXC1 fd, index(base) MIPS IV

Purpose: To load a word from memory to an FPR (GPR+GPR addressing).

Description: fd ← memory[base+index]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of coprocessor 1 general register *fd*. The contents of GPR *index* and GPR *base* are added to form the effective address.

If coprocessor 1 general registers are 64-bits wide, bits 63..32 of register *fd* become undefined. See Floating-Point Registers on page B-6.

Restrictions:

The Region bits of the effective address must be supplied by the contents of *base*. If $\text{EffectiveAddress}_{63..62} \neq base_{63..62}$, the result is undefined.

An Address Error exception occurs if $\text{EffectiveAddress}_{1..0} \neq 0$ (not word-aligned).

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation:

```
vAddr ← GPR[base] + GPR[index]
if vAddr_{1..0} ≠ 0^2 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_{PSIZE-1..3} || (pAddr_{2..0} xor (ReverseEndian || 0^2))
/* "mem" is aligned 64-bits from memory. Pick out correct bytes. */
mem ← LoadMemory(uncached, WORD, pAddr, vAddr, DATA)
bytesel ← vAddr_{2..0} xor (BigEndianCPU || 0^2)
if SizeFGR() = 64 then                                  /* 64-bit wide FGRs */
    FGR[fd] ← undefined^32 || mem_{31+8*bytesel..8*bytesel}
else                                                    /* 32-bit wide FGRs */
    FGR[fd] ← mem_{31+8*bytesel..8*bytesel}
endif
```

Exceptions:

TLB Refill, TLB Invalid
Address Error
Reserved Instruction
Coprocessor Unusable

31–26	25–21	20–16	15–11	10–6	5–3	2–0
COP1X 0 1 0 0 1 1	fr	ft	fs	fd	MADD 1 0 0	fmt
6	5	5	5	5	3	3

Format: MADD.S fd, fr, fs, ft **MIPS IV**
MADD.D fd, fr, fs, ft

Purpose: To perform a combined multiply-then-add of FP values.

Description: fd ← (fs × ft) + fr

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce a product. The value in FPR *fr* is added to the product. The result sum is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR *fd*. The operands and result are values in format *fmt*.

The accuracy of the result depends which of two alternative arithmetic models is used by the implementation for the computation. The numeric models are explained in Arithmetic Instructions on page B-21.

Restrictions:

The fields *fr, fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format *fmt*; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

```
vfr  ← ValueFPR(fr, fmt)
vfs  ← ValueFPR(fs, fmt)
vft  ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, vfr + vfs * vft)
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Inexact
- Unimplemented Operation
- Invalid Operation
- Overflow
- Underflow

MFC1 **Move Word From Floating-Point**

31–26	25–21	20–16	15–11	10–0
COP1 0 1 0 0 0 1	MF 0 0 0 0 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format: MFC1 rt, fs **MIPS I**

Purpose: To copy a word from an FPU (CP1) general register to a GPR.

Description: rt ← fs

The low word from FPR *fs* is placed into the low word of GPR *rt*. If GPR *rt* is 64 bits wide, then the value is sign extended. See Floating-Point Registers on page B-6.

Restrictions:

For MIPS I, MIPS II, and MIPS III the contents of GPR *rt* are undefined for the instruction immediately following MFC1.

Operation: MIPS I - III

I: word ← $FGR[fs]_{31..0}$
I+ 1 :GPR[rt] ← sign_extend(word)

Operation: MIPS IV

word ← $FGR[fs]_{31..0}$
GPR[rt]← sign_extend(word)

Exceptions:

Coprocessor Unusable

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	MOV 0 0 0 1 1 0
6	5	5	5	5	6

Format: MOV.S fd, fs **MIPS I**
MOV.D fd, fs

Purpose: To move an FP value between FPRs.

Description: fd ← fs

The value in FPR *fs* is placed into FPR *fd*. The source and destination are values in format *fmt*.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

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

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Unimplemented Operation

MOVF

Move Conditional on FP False

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

Format: MOVF rd, rs, cc **MIPS IV**

Purpose: To test an FP condition code then conditionally move a GPR.

Description: if (cc = 0) then rd ← rs

If the floating-point condition code specified by *cc* is zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

None

Operation:

```
active ← FCC[cc] = tf
if active then
    GPR[rd] ← GPR[rs]
endif
```

Exceptions:

Reserved Instruction
Coprocessor Unusable

Floating-Point Move Conditional on FP False **MOVF.fmt**

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

Format: MOVF.S fd, fs, cc **MIPS IV**
MOVF.D fd, fs, cc

Purpose: To test an FP condition code then conditionally move an FP value.

Description: if (cc = 0) then fd ← fs

If the floating-point condition code specified by *cc* is zero, then the value in FPR *fs* is placed into FPR *fd*. The source and destination are values in format *fmt*.

If the condition code is not zero, then FPR *fs* is not copied and FPR *fd* contains its previous value in format *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes undefined.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
if FCC[cc] = tf then
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Unimplemented operation

MOVN.fmt

Floating-Point Move Conditional on Not Zero

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	rt	fs	fd	MOVN 0 1 0 0 1 1
6	5	5	5	5	6

Format: MOVN.S fd, fs, rt
MOVN.D fd, fs, rt

MIPS IV

Purpose: To test a GPR then conditionally move an FP value.

Description: if (rt ≠ 0) then fd ← fs

If the value in GPR *rt* is not equal to zero then the value in FPR *fs* is placed in FPR *fd*. The source and destination are values in format *fmt*.

If GPR *rt* contains zero, then FPR *fs* is not copied and FPR *fd* contains its previous value in format *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes undefined.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
if GPR[rt] ≠ 0 then
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Unimplemented operation

Move Conditional on FP True **MOVT**

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

Format: MOVT rd, rs, cc **MIPS IV**

Purpose: To test an FP condition code then conditionally move a GPR.

Description: if (cc = 1) then rd ← rs

If the floating-point condition code specified by *cc* is one then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

None

Operation:

```
if FCC[cc] = tf then
    GPR[rd] ← GPR[rs]
endif
```

Exceptions:

Reserved Instruction
Coprocessor Unusable

MOVT.fmt

Floating-Point Move Conditional on FP True

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

Format: MOVT.S fd, fs, cc
MOVT.D fd, fs, cc

MIPS IV

Purpose: To test an FP condition code then conditionally move an FP value.

Description: if (cc = 1) then fd ← fs

If the floating-point condition code specified by *cc* is one then the value in FPR *fs* is placed into FPR *fd*. The source and destination are values in format *fmt*.

If the condition code is not one, then FPR *fs* is not copied and FPR *fd* contains its previous value in format *fm*t. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes undefined.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
if FCC[cc] = tf then
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Unimplemented operation

Floating-Point Move Conditional on Zero **MOVZ.fmt**

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	rt	fs	fd	MOVZ 0 1 0 0 1 0
6	5	5	5	5	6

Format: MOVZ.S fd, fs, rt MIPS IV
MOVZ.D fd, fs, rt

Purpose: To test a GPR then conditionally move an FP value.

Description: if (rt = 0) then fd ← fs

If the value in GPR *rt* is equal to zero then the value in FPR *fs* is placed in FPR *fd*. The source and destination are values in format *fmt*.

If GPR *rt* is not zero, then FPR *fs* is not copied and FPR *fd* contains its previous value in format *fm*t. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes undefined.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
if GPR[rt] = 0 then
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Unimplemented operation

MSUB.fmt

Floating-Point Multiply Subtract

31–26	25–21	20–16	15–11	10–6	5–3	2–0
COP1X 010011	fr	ft	fs	fd	MSUB 101	fmt
6	5	5	5	5	3	3

Format: MSUB.S fd, fr, fs, ft **MIPS IV**
MSUB.D fd, fr, fs, ft

Purpose: To perform a combined multiply-then-subtract of FP values.

Description: fd ← (fs × ft) - fr

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce an intermediate product. The value in FPR *fr* is subtracted from the product. The subtraction result is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR *fd*. The operands and result are values in format *fmt*.

The accuracy of the result depends which of two alternative arithmetic models is used by the implementation for the computation. The numeric models are explained in Arithmetic Instructions on page B-21.

Restrictions:

The fields *fr*, *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format *fmt*; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

```
vfr  ← ValueFPR(fr, fmt)
vfs  ← ValueFPR(fs, fmt)
vft  ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, (vfs * vft) - vfr)
```

Exceptions:

Reserved Instruction
Coprocessor Unusable
Floating-Point
- Inexact
- Invalid Operation
- Underflow
- Unimplemented Operation
- Overflow

Move Word to Floating-Point **MTC1**

31–26	25–21	20–16	15–11	10–0
COP1 0 1 0 0 0 1	MT 0 0 1 0 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format: MTC1 rt, fs **MIPS I**

Purpose: To copy a word from a GPR to an FPU (CP1) general register.

Description: fs ← rt

The low word in GPR *rt* is placed into the low word of floating-point (coprocessor 1) general register *fs*. If coprocessor 1 general registers are 64-bits wide, bits 63..32 of register *fs* become undefined. See Floating-Point Registers on page B-6.

Restrictions:

For MIPS I, MIPS II, and MIPS III the value of FPR *fs* is undefined for the instruction immediately following MTC1.

Operation: MIPS I - III

```
I:      data ← GPR[rt]31..0
I+1:    if SizeFGR() = 64 then                 /* 64-bit wide FGRs */
            FGR[fs] ← undefined^32 || data
        else                                   /* 32-bit wide FGRs */
            FGR[fs] ← data
        endif
```

Operation: MIPS IV

```
        data ← GPR[rt]31..0
        if SizeFGR() = 64 then                 /* 64-bit wide FGRs */
            FGR[fs] ← undefined^32 || data
        else                                   /* 32-bit wide FGRs */
            FGR[fs] ← data
        endif
```

Exceptions:

Coprocessor Unusable

MUL.fmt

Floating-Point Multiply

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	ft	fs	fd	MUL 0 0 0 0 1 0
6	5	5	5	5	6

Format: MUL.S fd, fs, ft
MUL.D fd, fs, ft

MIPS I

Purpose: To multiply FP values.

Description: fd ← fs × ft

The value in FPR *fs* is multiplied by the value in FPR *ft*. The result is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR *fd*. The operands and result are values in format *fmt*.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format *fmt*; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

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

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Inexact
- Invalid Operation
- Underflow
- Unimplemented Operation
- Overflow

Floating-Point Negate **NEG.fmt**

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	NEG 0 0 0 1 1 1
6	5	5	5	5	6

Format: NEG.S fd, fs **MIPS I**
NEG.D fd, fs

Purpose: To negate an FP value.

Description: fd ← - (fs)

The value in FPR *fs* is negated and placed into FPR *fd*. The value is negated by changing the sign bit value. The operand and result are values in format *fmt*.

This operation is arithmetic; a NaN operand signals invalid operation.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

StoreFPR(fd, fmt, Negate(ValueFPR(fs, fmt)))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Unimplemented Operation
  Invalid Operation

NMADD.fmt

Floating-Point Negative Multiply Add

31–26	25–21	20–16	15–11	10–6	5–3	2–0
COP1X 0 1 0 0 1 1	fr	ft	fs	fd	NMADD 1 1 0	fmt
6	5	5	5	5	3	3

Format: NMADD.S fd, fr, fs, ft
NMADD.D fd, fr, fs, ft

MIPS IV

Purpose: To negate a combined multiply-then-add of FP values.

Description: fd $\leftarrow$ - ((fs $\times$ ft) + fr)

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce an intermediate product. The value in FPR *fr* is added to the product. The result sum is calculated to infinite precision, rounded according to the current rounding mode in FCSR, negated by changing the sign bit, and placed into FPR *fd*. The operands and result are values in format *fmt*.

The accuracy of the result depends which of two alternative arithmetic models is used by the implementation for the computation. The numeric models are explained in Arithmetic Instructions on page B-21.

Restrictions:

The fields *fr, fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format *fmt*; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, -(vfr + vfs * vft))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Inexact
- Invalid Operation
- Underflow
- Unimplemented Operation
- Overflow

NMSUB.fmt

31–26	25–21	20–16	15–11	10–6	5–3	2–0
COP1X 0 1 0 0 1 1	fr	ft	fs	fd	NMSUB 1 1 1	fmt
6	5	5	5	5	3	3

Format: NMSUB.S fd, fr, fs, ft **MIPS IV**
NMSUB.D fd, fr, fs, ft

Purpose: To negate a combined multiply-then-subtract of FP values.

Description: fd ← - ((fs × ft) - fr)

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce an intermediate product. The value in FPR *fr* is subtracted from the product. The result is calculated to infinite precision, rounded according to the current rounding mode in FCSR, negated by changing the sign bit, and placed into FPR *fd*. The operands and result are values in format *fmt*.

The accuracy of the result depends which of two alternative arithmetic models is used by the implementation for the computation. The numeric models are explained in Arithmetic Instructions on page B-21.

Restrictions:

The fields *fr*, *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format *fmt*; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

```
vfr  ← ValueFPR(fr, fmt)
vfs  ← ValueFPR(fs, fmt)
vft  ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, -((vfs * vft) - vfr))
```

Exceptions:

Reserved Instruction
Coprocessor Unusable
Floating-Point
- Inexact
- Invalid Operation
- Underflow
- Unimplemented Operation
- Overflow

PREFX

Prefetch Indexed

31–26	25–21	20–16	15–11	10–6	5–0
COP1X 0 1 0 0 1 1	base	index	hint	0 0 0 0 0 0	PREFX 0 0 1 1 1 1
6	5	5	5	5	6

Format: PREFX hint, index(base) MIPS IV

Purpose: To prefetch locations from memory (GPR+GPR addressing).

Description: prefetch_memory[base+index]

PREFX adds the contents of GPR *index* to the contents of GPR *base* to form an effective byte address. It advises that data at the effective address may be used in the near future. The *hint* field supplies information about the way that the data is expected to be used.

PREFX is an advisory instruction. It may change the performance of the program. For all *hint* values, it neither changes architecturally-visible state nor alters the meaning of the program. An implementation may do nothing when executing a PREFX instruction.

If MIPS IV instructions are supported and enabled and Coprocessor 1 is enabled (allowing access to CP1X), PREFX does not cause addressing-related exceptions. If it raises an exception condition, the exception condition is ignored. If an addressing-related exception condition is raised and ignored, no data will be prefetched. Even if no data is prefetched in such a case, some action that is not architecturally-visible, such as writeback of a dirty cache line, might take place.

PREFX will never generate a memory operation for a location with an uncached memory access type (see Memory Access Types on page A-12).

If PREFX results in a memory operation, the memory access type used for the operation is determined by the memory access type of the effective address, just as it would be if the memory operation had been caused by a load or store to the effective address.

PREFX enables the processor to take some action, typically prefetching the data into cache, to improve program performance. The action taken for a specific PREFX instruction is both system and context dependent. Any action, including doing nothing, is permitted that does not change architecturally-visible state or alter the meaning of a program. It is expected that implementations will either do nothing or take an action that will increase the performance of the program.

For a cached location, the expected, and useful, action is for the processor to prefetch a block of data that includes the effective address. The size of the block, and the level of the memory hierarchy it is fetched into are implementation specific.

The *hint* field supplies information about the way the data is expected to be used. No *hint* value causes an action that modifies architecturally-visible state. A processor may use a *hint* value to improve the effectiveness of the prefetch action. The defined *hint* values and the recommended prefetch action are shown in the table below. The *hint* table may be extended in future implementations.

Table B-22 Values of Hint Field for Prefetch Instruction

Value	Name	Data use and desired prefetch action
0	load	Data is expected to be loaded (not modified). Fetch data as if for a load.
1	store	Data is expected to be stored or modified. Fetch data as if for a store.
2-3		Not yet defined.
4	load_streamed	Data is expected to be loaded (not modified) but not reused extensively; it will “stream” through cache. Fetch data as if for a load and place it in the cache so that it will not displace data prefetched as “retained”.
5	store_streamed	Data is expected to be stored or modified but not reused extensively; it will “stream” through cache. Fetch data as if for a store and place it in the cache so that it will not displace data prefetched as “retained”.
6	load_retained	Data is expected to be loaded (not modified) and reused extensively; it should be “retained” in the cache. Fetch data as if for a load and place it in the cache so that it will not be displaced by data prefetched as “streamed”.
7	store_retained	Data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Fetch data as if for a store and place it in the cache so that will not be displaced by data prefetched as “streamed”.
8-31		Not yet defined.

Restrictions:

The Region bits of the effective address must be supplied by the contents of *base*. If $\text{EffectiveAddress}_{63..62} \neq base_{63..62}$, the result of the instruction is undefined.

Operation:

```
vAddr ← GPR[base] + GPR[index]
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(uncached, pAddr, vAddr, DATA, hint)
```

Exceptions:

Reserved Instruction
Coprocessor Unusable

Programming Notes:

Prefetch can not prefetch data from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. It will not cause an exception to prefetch using an address pointer value before the validity of a pointer is determined.

Implementation Notes:

It is recommended that a reserved *hint* field value either cause a default prefetch action that is expected to be useful for most cases of data use, such as the "load" *hint*, or cause the instruction to be treated as a NOP.

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	RECIP 0 1 0 1 0 1
6	5	5	5	5	6

Format: RECIP.S fd, fs **MIPS IV**
RECIP.D fd, fs

Purpose: To approximate the reciprocal of an FP value (quickly).

Description: fd ← 1.0 / fs

The reciprocal of the value in FPR *fs* is approximated and placed into FPR *fd*. The operand and result are values in format *fmt*.

The numeric accuracy of this operation is implementation dependent; it does not meet the accuracy specified by the IEEE 754 Floating-Point standard. The computed result differs from the both the exact result and the IEEE-mandated representation of the exact result by no more than one unit in the least-significant place (ulp).

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

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
StoreFPR(fd, fmt, 1.0 / valueFPR(fs, fmt))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point

Inexact	Unimplemented Operation
Division-by-zero	Invalid Operation
Overflow	Underflow

ROUND.L.fmt — Floating-Point Round to Long Fixed-Point

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	ROUND.L 0 0 1 0 0 0
6	5	5	5	5	6

Format: ROUND.L.S fd, fs — MIPS III
ROUND.L.D fd, fs

Purpose: To convert an FP value to 64-bit fixed-point, rounding to nearest.

Description: fd ← convert_and_round(fs)

The value in FPR *fs* in format *fmt*, is converted to a value in 64-bit long fixed-point format rounding to nearest/even (rounding mode 0). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: The Invalid Operation flag is set in the FCSR. If the Invalid Operation enable bit is set in the FCSR, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}–1$, is written to *fd*.
- Imprecise exception model (R8000 normal mode): The default result, $2^{63}–1$, is written to *fd*. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Inexact
- Overflow
- Unimplemented Operation
- Invalid Operation

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	ROUND.W 0 0 1 1 0 0
6	5	5	5	5	6

Format: ROUND.W.S fd, fs
ROUND.W.D fd, fs

MIPS II

Purpose: To convert an FP value to 32-bit fixed-point, rounding to nearest.

Description: fd ← convert_and_round(fs)

The value in FPR *fs* in format *fmt*, is converted to a value in 32-bit word fixed-point format rounding to nearest/even (rounding mode 0). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: The Invalid Operation flag is set in the FCSR. If the Invalid Operation enable bit is set in the FCSR, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}–1$, is written to *fd*.
- Imprecise exception model (R8000 normal mode): The default result, $2^{31}–1$, is written to *fd*. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
Inexact — Unimplemented Operation
Invalid Operation — Overflow

RSQRT.fmt Reciprocal Square Root Approximation

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	RSQRT 0 1 0 1 1 0
6	5	5	5	5	6

Format: RSQRT.S fd, fs **MIPS IV**
RSQRT.D fd, fs

Purpose: To approximate the reciprocal of the square root of an FP value (quickly).

Description: fd ← 1.0 / sqrt(fs)

The reciprocal of the positive square root of the value in FPR *fs* is approximated and placed into FPR *fd*. The operand and result are values in format *fmt*.

The numeric accuracy of this operation is implementation dependent; it does not meet the accuracy specified by the IEEE 754 Floating-Point standard. The computed result differs from the both the exact result and the IEEE-mandated representation of the exact result by no more than two units in the least-significant place (ulp).

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

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
StoreFPR(fd, fmt, 1.0 / SquareRoot(valueFPR(fs, fmt)))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point

Inexact	Unimplemented Operation
Division-by-zero	Invalid Operation
Overflow	Underflow

31–26	25–21	20–16	15–0
SDC1 1 1 1 1 0 1	base	ft	offset
6	5	5	16

Format: SDC1 ft, offset(base) **MIPS II**

Purpose: To store a doubleword from an FPR to memory.

Description: memory[base+offset] ← ft

The 64-bit doubleword in FPR *ft* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR *ft* is held in an even/odd register pair. The low word is taken from the even register *ft* and the high word is from *ft*+1.

Restrictions:

If *ft* does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Registers on page B-6.

An Address Error exception occurs if $\text{EffectiveAddress}_{2..0} \neq 0$ (not doubleword-aligned).

MIPS IV: The low-order 3 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, STORE)
if SizeFGR() = 64 then                /* 64-bit wide FGRs */
    data ← FGR[ft]
elseif ft0 = 0 then                   /* valid specifier, 32-bit wide FGRs */
    data ← FGR[ft+1] || FGR[ft]
else                                  /* undefined for odd 32-bit FGRs */
    UndefinedResult()
endif
StoreMemory(uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor unusable
Reserved Instruction
TLB Refill, TLB Invalid
TLB Modified
Address Error

SDXC1 Store Doubleword Indexed from Floating-Point

31–26	25–21	20–16	15–11	10–6	5–0
COP1X 0 1 0 0 1 1	base	index	fs	0	SDXC1 0 0 1 0 0 1
6	5	5	5	5	6

Format: SDXC1 fs, index(base) **MIPS IV**

Purpose: To store a doubleword from an FPR to memory (GPR+GPR addressing).

Description: memory[base+index] ← fs

The 64-bit doubleword in FPR *fs* is stored in memory at the location specified by the aligned effective address. The contents of GPR *index* and GPR *base* are added to form the effective address.

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR *fs* is held in an even/odd register pair. The low word is taken from the even register *fs* and the high word is from *fs*+1.

Restrictions:

If *fs* does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Registers on page B-6.

The Region bits of the effective address must be supplied by the contents of *base*. If $\mathrm{EffectiveAddress}_{63..62} \neq base_{63..62}$, the result is undefined.

An Address Error exception occurs if $\mathrm{EffectiveAddress}_{2..0} \neq 0$ (not doubleword-aligned).

MIPS IV: The low-order 3 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation:

```
vAddr ← GPR[base] + GPR[index]
if vAddr2..0 ≠ 0^3 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, STORE)
if SizeFGR() = 64 then                    /* 64-bit wide FGRs */
    data ← FGR[fs]
elseif fs0 = 0 then                       /* valid specifier, 32-bit wide FGRs */
    data ← FGR[fs+1] || FGR[fs]
else                                      /* undefined for odd 32-bit FGRs */
    UndefinedResult()
endif
StoreMemory(uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error
Reserved Instruction
Coprocessor Unusable

SQRT.fmt

Floating-Point Square Root

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	SQRT 0 0 0 1 0 0
6	5	5	5	5	6

Format: SQRT.S fd, fs **MIPS II**
SQRT.D fd, fs

Purpose: To compute the square root of an FP value.

Description: fd ← SQRT(fs)

The square root of the value in FPR *fs* is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR *fd*. The operand and result are values in format *fmt*.

If the value in FPR *fs* corresponds to –0, the result will be –0.

Restrictions:

If the value in FPR *fs* is less than 0, an Invalid Operation condition is raised.

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
StoreFPR(fd, fmt, SquareRoot(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Unimplemented Operation
- Invalid Operation
- Inexact

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	ft	fs	fd	SUB 0 0 0 0 0 1
6	5	5	5	5	6

Format: SUB.S fd, fs, ft **MIPS I**
SUB.D fd, fs, ft

Purpose: To subtract FP values.

Description: fd ← fs - ft

The value in FPR *ft* is subtracted from the value in FPR *fs*. The result is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR *fd*. The operands and result are values in format *fmt*.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operands must be values in format *fmt*; see section B 7 on page B-24. If they are not, the result is undefined and the value of the operand FPRs becomes undefined.

Operation:

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

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Inexact
- Unimplemented Operation
- Invalid Operation
- Overflow
- Underflow

SWC1 — Store Word from Floating-Point

31–26	25–21	20–16	15–0
SWC1 1 1 1 0 0 1	base	ft	offset
6	5	5	16

Format: SWC1 ft, offset(base) **MIPS I**

Purpose: To store a word from an FPR to memory.

Description: memory[base+offset] ← ft

The low 32-bit word from FPR *ft* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

An Address Error exception occurs if $\text{EffectiveAddress}_{1..0} \neq 0$ (not word-aligned).

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32-bit Processors

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
data ← FGR[ft]
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Operation: 64-bit Processors

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 | | (pAddr2..0 xor (ReverseEndian || 0^2))
bytesel ← vAddr2..0 xor (BigEndianCPU || 0^2)
/* the bytes of the word are moved into the correct byte lanes */
if SizeFGR() = 64 then                                  /* 64-bit wide FGRs */
    data ← 0^(32-8*bytesel) || FGR[ft]31..0 || 0^(8*bytesel) /* top or bottom wd of 64-bit data */
else                                                    /* 32-bit wide FGRs */
    data ← 0^(32-8*bytesel) || FGR[ft] || 0^(8*bytesel)      /* top or bottom wd of 64-bit data */
endif
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor unusable
Reserved Instruction
TLB Refill, TLB Invalid
TLB Modified
Address Error

Store Word Indexed from Floating-Point SWXC1

31–26	25–21	20–16	15–11	10–6	5–0
COP1X 0 1 0 0 1 1	base	index	fs	0	SWXC1 0 0 1 0 0 0
6	5	5	5	5	6

Format: SWXC1 fs, index(base) **MIPS IV**

Purpose: To store a word from an FPR to memory (GPR+GPR addressing).

Description: memory[base+index] ← fs

The low 32-bit word from FPR *fs* is stored in memory at the location specified by the aligned effective address. The contents of GPR *index* and GPR *base* are added to form the effective address.

Restrictions:

The Region bits of the effective address must be supplied by the contents of *base*. If $EffectiveAddress_{63..62} \neq base_{63..62}$, the result is undefined.

An Address Error exception occurs if $EffectiveAddress_{1..0} \neq 0$ (not word-aligned).

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation:

```
vAddr ← GPR[base] + GPR[index]
if vAddr1..0 ≠ 0^2 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
bytesel ← vAddr2..0 xor (BigEndianCPU || 0^2)
/* the bytes of the word are moved into the correct byte lanes */
if SizeFGR() = 64 then                                /* 64-bit wide FGRs */
    data ← 0^(32-8*bytesel) || FGR[fs]31..0 || 0^(8*bytesel) /* top or bottom wd of 64-bit data */
else                                                  /* 32-bit wide FGRs */
    data ← 0^(32-8*bytesel) || FGR[fs] || 0^(8*bytesel)      /* top or bottom wd of 64-bit data */
endif
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error
Reserved Instruction
Coprocessor Unusable

TRUNC.L.fmt

Floating-Point Truncate to Long Fixed-Point

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	TRUNC.L 0 0 1 0 01
6	5	5	5	5	6

Format: TRUNC.L.S fd, fs **MIPS III**
TRUNC.L.D fd, fs

Purpose: To convert an FP value to 64-bit fixed-point, rounding toward zero.

Description: fd ← convert_and_round(fs)

The value in FPR *fs* in format *fmt*, is converted to a value in 64-bit long fixed-point format rounding toward zero (rounding mode 1). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to 2^{63}-1, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: The Invalid Operation flag is set in the FCSR. If the Invalid Operation enable bit is set in the FCSR, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}–1$, is written to *fd*.
- Imprecise exception model (R8000 normal mode): The default result, $2^{63}–1$, is written to *fd*. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Inexact Unimplemented Operation
  Invalid Operation Overflow

Floating-Point Truncate to Word Fixed-Point **TRUNC.W.fmt**

31–26	25–21	20–16	15–11	10–6	5–0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	TRUNC.W 0 0 1 1 0 1
6	5	5	5	5	6

Format: TRUNC.W.S fd, fs **MIPS II**
TRUNC.W.D fd, fs

Purpose: To convert an FP value to 32-bit fixed-point, rounding toward zero.

Description: fd ← convert_and_round(fs)

The value in FPR *fs* in format *fmt*, is converted to a value in 32-bit word fixed-point format using rounding toward zero (rounding mode 1)). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to 2^{31}-1, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. The result depends on the FP exception model currently active.

- Precise exception model: The Invalid Operation flag is set in the FCSR. If the Invalid Operation enable bit is set in the FCSR, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}–1$, is written to *fd*.
- Imprecise exception model (R8000 normal mode): The default result, $2^{31}–1$, is written to *fd*. No FCSR flag is set. If the Invalid Operation enable bit is set in the FCSR, an Invalid Operation exception is taken, imprecisely, at some future time.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed-point; see Floating-Point Registers on page B-6. If they are not valid, the result is undefined.

The operand must be a value in format *fmt*; see section B 7 on page B-24. If it is not, the result is undefined and the value of the operand FPR becomes undefined.

Operation:

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Inexact   Invalid Operation
  Overflow   Unimplemented Operation

B.11 FPU Instruction Formats

An FPU instruction is a single 32-bit aligned word. The distinct FP instruction layouts are shown in Figure B-16. Variable information is in lower-case labels, such as “offset”. Upper-case labels and any numbers indicate constant data. A table follows all the layouts that explains the fields used in them. Note that the same field may have different names in different instruction layout pictures. The field name is mnemonic to the function of that field in the instruction layout. The opcode tables and the instruction decode discussion use the canonical field names: opcode, fmt, nd, tf, and function. The other fields are not used for instruction decode.

Figure B-16 FPU Instruction Formats

Immediate: load/store using register + offset addressing.

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

Register: 2-register and 3-register formatted arithmetic operations.

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

Register Immediate: data transfer -- CPU ↔ FPU register.

31 26	25 21	20 16	15 11	10 0
COP1	sub	rt	fs	0
6	5	5	5	11

Condition code, Immediate: conditional branches on FPU cc using PC + offset.

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

Register to Condition Code: formatted FP compare.

31 26	25 21	20 16	15 11	10 8	7 6	5 0
COP1	fmt	ft	fs	cc	0	function
6	5	5	5	3	2	4

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

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

Register-4: 4-register formatted arithmetic operations.

31 26	25 21	20 16	15 11	10 6	5 3	2 0
					function	
COP1X	fr	ft	fs	fd	op4	fmt3
6	5	5	5	5	3	3

Register Index: Load/store using register + register addressing.

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

Register Index hint: Prefetch using register + register addressing.

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

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

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

BC	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 architecture levels prior to MIPS IV it must be zero.
fd	FPU register: destination (arithmetic, loads, move-to) or source (stores, move-from)
fmt	destination and/or operand type ("format") specifier
fr	FPU register: source
fs	FPU register: source
ft	FPU register: source (for stores, arithmetic) or destination (for loads)
function	function field specifying a function within a particular op operation code.
function: op4 + fmt3	op4 is a 3-bit function field specifying which 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 several distinct instructions in the opcode tables.
hint	hint field made available to cache controller for prefetch operation
index	CPU register, holds index address component for address calculations
MOVC	Value in function field for conditional move. There is one value for the instruction with op=COP1, another for the instruction with op=SPECIAL.
nd	nullify delay. If set, branch is Likely and delay slot instruction is not executed. This must be zero for MIPS I.
offset	signed offset field used in address calculations
op	primary operation code (COP1, COP1X, LWC1, SWC1, LDC1, SDC1, SPECIAL)
PREFX	Value in function field for prefetch instruction for op=COP1X
rd	CPU register: destination
rs	CPU register: source
rt	CPU register: source / 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 FP compare is tested for equality with tf bit.

B.12 FPU (CP1) Instruction Opcode Bit Encoding

This section describes the encoding of the Floating-Point Unit (FPU) instructions for the four levels of the MIPS architecture, MIPS I through MIPS IV. Each architecture level includes the instructions in the previous level;† MIPS IV includes all instructions in MIPS I, MIPS II, and MIPS III. This section presents eight different views of the instruction encoding.

- Separate encoding tables for each architecture level.
- A MIPS IV encoding table showing the architecture level at which each opcode was originally defined and subsequently modified (if modified).
- Separate encoding tables for each architecture revision showing the changes made during that revision.

B 12.1 Instruction Decode

Instruction field names are printed in **bold** in this section.

The primary **opcode** field is decoded first. The **opcode** values LWC1, SWC1, LDC1, and SDC1 fully specify FPU load and store instructions. The **opcode** values *COP1*, *COP1X*, and *SPECIAL* specify instruction classes. Instructions within a class are further specified by values in other fields.

B 12.1.1 COP1 Instruction Class

The **opcode** =*COP1* instruction class encodes most of the FPU instructions. The class is further decoded by examining the **fmt** field. The **fmt** values fully specify the CPU ↔ FPU register move instructions and specify the *S*, *D*, *W*, *L*, and *BC* instruction classes.

The **opcode** =*COP1* + **fmt** =*BC* instruction class encodes the conditional branch instructions. The class is further decoded, and the instructions fully specified, by examining the **nd** and **tf** fields.

The **opcode** =*COP1* + **fmt** =(*S*, *D*, *W*, or *L*) instruction classes encode instructions that operate on formatted (typed) operands. Each of these instruction classes is further decoded by examining the **function** field. With one exception the **function** values fully specify instructions. The exception is the *MOVCF* instruction class.

The **opcode** =*COP1* + **fmt** =(*S* or *D*) + **function** =*MOVCF* instruction class encodes the MOVT.*fmt* and MOVF.*fmt* conditional move instructions (to move FP values based on FP condition codes). The class is further decoded, and the instructions fully specified, by examining the **tf** field.

† An exception to this rule is that the reserved, but never implemented, Coprocessor 3 instructions were removed or changed to another use starting in MIPS III.

B 12.1.2 COP1X Instruction Class

The opcode =*COP1X* instruction class encodes the indexed load/store instructions, the indexed prefetch, and the multiply accumulate instructions. The class is further decoded, and the instructions fully specified, by examining the function field.

B 12.1.3 SPECIAL Instruction Class

The opcode =*SPECIAL* instruction class is further decoded by examining the function field. The only function value that applies to FPU instruction encoding is the *MOVCI* instruction class. The remainder of the function values encode CPU instructions.

The opcode =*SPECIAL* + function =*MOVCI* instruction class encodes the MOVT and MOVF conditional move instructions (to move CPU registers based on FP condition codes). The class is further decoded, and the instructions fully specified, by examining the tf field.

B 12.2 Instruction Subsets of MIPS III and MIPS IV Processors.

MIPS III processors, such as the R4000, R4200, R4300, R4400, and R4600, have a processor mode in which only the MIPS II instructions are valid. The MIPS II encoding table describes the MIPS II-only mode.

MIPS IV processors, such as the R8000 and R10000, have processor modes in which only the MIPS II or MIPS III instructions are valid. The MIPS II encoding table describes the MIPS II-only mode. The MIPS III encoding table describes the MIPS III-only mode.

Table B-23 FPU (CP1) Instruction Encoding - MIPS I Architecture

Instructions encoded by the opcode field.

31 26	0
opcode	

opcode bits 31..29	bits 28..26: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001								
2 010		*COP1* δ						
3 011								
4 100					χ			
5 101								
6 110		LWC1						
7 111		SWC1						

Instructions encoded by the fmt field when opcode=*COP1*.

31 26	25 21	0
opcode = *COP1*	**fmt**	

fmt bits 25..24	bits 23..21: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00	MFC1	*	CFC1	*	MTC1	*	CTC1	*
1 01	*BC* δ	*	*	*	*	*	*	*
2 10	*S* δ	*D* δ	*	*	*W* δ	*	*	*
3 11	*	*	*	*	*	*	*	*

Instructions encoded by the tf field when opcode=*COP1* and fmt=*BC*.

31 26	25 21		16	0
opcode = *COP1*	fmt = *BC*		**tf**	

tf bit 16: 0	1
BC1F	BC1T

Table B-23 (cont.) FPU (CP1) Instruction Encoding - MIPS I Architecture

Instructions encoded by the function field when opcode=*COP1* and fmt = *S*, *D*, or *W*

encoding when fmt = *S*

31–26	25–21		0
opcode = *COP1*	fmt = *S*		**function**

function bits 5..3		bits 2..0: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	ADD	SUB	MUL	DIV	*	ABS	MOV	NEG
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	*	CVT.D	*	*	CVT.W	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F α	C.UN α	C.EQ α	C.UEQ α	C.OLT α	C.ULT α	C.OLE α	C.ULE α
7	111	C.SF α	C.NGLE α	C.SEQ α	C.NGL α	C.LT α	C.NGE α	C.LE α	C.NGT α

encoding when fmt = *D*

31–26	25–21		0
opcode = *COP1*	fmt = *D*		**function**

function bits 5..3		bits 2..0: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	ADD	SUB	MUL	DIV	*	ABS	MOV	NEG
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	CVT.S	*	*	*	CVT.W	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F α	C.UN α	C.EQ α	C.UEQ α	C.OLT α	C.ULT α	C.OLE α	C.ULE α
7	111	C.SF α	C.NGLE α	C.SEQ α	C.NGL α	C.LT α	C.NGE α	C.LE α	C.NGT α

encoding when
fmt = *W*

31–26	25–21		0
opcode = *COP1*	fmt = *W*		**function**

function bits 5..3	bits 2..0: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	*	*	*	*	*	*	*	*
1 001	*	*	*	*	*	*	*	*
2 010	*	*	*	*	*	*	*	*
3 011	*	*	*	*	*	*	*	*
4 100	CVT.S	CVT.D	*	*	*	*	*	*
5 101	*	*	*	*	*	*	*	*
6 110	*	*	*	*	*	*	*	*
7 111	*	*	*	*	*	*	*	*

Table B-24 FPU (CP1) Instruction Encoding - MIPS II Architecture

Instructions encoded by the opcode field.

31 26	0
opcode	

opcode bits 31..29		bits 28..26 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000								
1	001								
2	010		*COP1* δ						
3	011								
4	100				χ				
5	101								
6	110		LWC1				LDC1		
7	111		SWC1				SDC1		

Instructions encoded by the fmt field when opcode=*COP1*.

31 26	25 21	0
opcode = *COP1*	**fmt**	

fmt bits 25..24		bits 23..21 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	00	MFC1	*	CFC1	*	MTC1	*	CTC1	*
1	01	*BC* δ	*	*	*	*	*	*	*
2	10	*S* δ	*D* δ	*	*	*W* δ	*	*	*
3	11	*	*	*	*	*	*	*	*

Instructions encoded by the nd and tf fields when opcode=*COP1* and fmt=*BC*.

31 26	25 21		17	16	0
opcode = *COP1*	fmt = *BC*		**nd**	**tf**	

nd bit 17	tf bit 16 0	1
0	BC1F	BC1T
1	BC1FL	BC1TL

Table B-24 (cont.) FPU (CP1) Instruction Encoding - MIPS II Architecture

Instructions encoded by the function field when opcode=*COP1* and fmt = *S*, *D*, or *W*

encoding when fmt = *S*

31–26	25–21		0
opcode = *COP1*	fmt = *S*		**function**

function bits 5..3 \ bits 2..0		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	*	*	*	*	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	*	CVT.D	*	*	CVT.W	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F α	C.UN α	C.EQ α	C.UEQ α	C.OLT α	C.ULT α	C.OLE α	C.ULE α
7	111	C.SF α	C.NGLE α	C.SEQ α	C.NGL α	C.LT α	C.NGE α	C.LE α	C.NGT α

encoding when fmt = *D*

31–26	25–21		0
opcode = *COP1*	fmt = *D*		**function**

function bits 5..3 \ bits 2..0		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	*	*	*	*	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	CVT.S	*	*	*	CVT.W	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F α	C.UN α	C.EQ α	C.UEQ α	C.OLT α	C.ULT α	C.OLE α	C.ULE α
7	111	C.SF α	C.NGLE α	C.SEQ α	C.NGL α	C.LT α	C.NGE α	C.LE α	C.NGT α

encoding when fmt = *W*

31–26	25–21		0
opcode = *COP1*	fmt = *W*		**function**

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

Table B-25 FPU (CP1) Instruction Encoding - MIPS III Architecture

Instructions encoded by the opcode field.

31..26	25..0
opcode	

opcode bits 31..29	bits 28..26: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001								
2 010		*COP1* δ						
3 011								
4 100				χ				
5 101								
6 110		LWC1				LDC1		
7 111		SWC1				SDC1		

Instructions encoded by the fmt field when opcode=*COP1*.

31..26	25..21	20..0
opcode = *COP1*	**fmt**	

fmt bits 25..24	bits 23..21: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00	MFC1	DMFC1	CFC1	*	MTC1	DMTC1	CTC1	*
1 01	*BC* δ	*	*	*	*	*	*	*
2 10	*S* δ	*D* δ	*	*	*W* δ	*L* δ	*	*
3 11	*	*	*	*	*	*	*	*

Instructions encoded by the nd and tf fields when opcode=*COP1* and fmt=*BC*.

31..26	25..21	20..18	17	16	15..0
opcode = *COP1*	fmt = *BC*		**nd**	**tf**	

nd bit 17	tf bit 16: 0	1
0	BC1F	BC1T
1	BC1FL	BC1TL

Table B-25 (cont.) FPU (CP1) Instruction Encoding - MIPS III Architecture

Instructions encoded by the function field when opcode=*COP1* and fmt = *S*, *D*, *W*, or *L*

encoding when fmt = *S*

31 26	25 21		0
opcode = *COP1*	fmt = *S*		**function**

function bits 5..3		bits 2..0 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L	TRUNC.L	CEIL.L	FLOOR.L	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	*	*	*	*	*	*	
3	011	*	*	*	*	*	*	*	*
4	100	*	CVT.D	*	*	CVT.W	CVT.L	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F α	C.UN α	C.EQ α	C.UEQ α	C.OLT α	C.ULT α	C.OLE α	C.ULE α
7	111	C.SF α	C.NGLE α	C.SEQ α	C.NGL α	C.LT α	C.NGE α	C.LE α	C.NGT α

encoding when fmt = *D*

31 26	25 21		0
opcode = *COP1*	fmt = *D*		**function**

function bits 5..3		bits 2..0 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L	TRUNC.L	CEIL.L	FLOOR.L	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	*	*	*	*	*	*	
3	011	*	*	*	*	*	*	*	*
4	100	CVT.S	*	*	*	CVT.W	CVT.L	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F α	C.UN α	C.EQ α	C.UEQ α	C.OLT α	C.ULT α	C.OLE α	C.ULE α
7	111	C.SF α	C.NGLE α	C.SEQ α	C.NGL α	C.LT α	C.NGE α	C.LE α	C.NGT α

encoding when fmt = *W* or *L*

31–26	25–21		0
opcode = *COP1*	fmt = *W, L*		**function**

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

Table B-26 FPU (CP1) Instruction Encoding - MIPS IV Architecture

Instructions encoded by the opcode field.

31..26	25..0
opcode	

opcode bits 31..29		bits 28..26: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	*SPECIAL* δ, β							
1	001								
2	010		*COP1* δ		*COP1X* δ,λ		χ		
3	011								
4	100								
5	101								
6	110		LWC1				LDC1		
7	111		SWC1				SDC1		

Instructions encoded by the fmt field when opcode=*COP1*.

31..26	25..21	20..0
opcode = *COP1*	**fmt**	

fmt bits 25..24		bits 23..21: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	00	MFC1	DMFC1	CFC1	*	MTC1	DMTC1	CTC1	*
1	01	*BC* δ	*	*	*	*	*	*	*
2	10	*S* δ	*D* δ	*	*	*W* δ	*L* δ	*	*
3	11	*	*	*	*	*	*	*	*

Instructions encoded by the nd and tf fields when opcode=*COP1* and fmt=*BC*.

31..26	25..21		17	16	15..0
opcode = *COP1*	fmt = *BC*		**nd**	**tf**	

nd bit 17	tf bit 16: 0	1
0	BC1F	BC1T
1	BC1FL	BC1TL

Table B-26 (cont.) FPU (CP1) Instruction Encoding - MIPS IV Architecture

Instructions encoded by the function field when opcode=*COP1* and fmt = *S*, *D*, *W*, or *L*

encoding when fmt = *S*

31 ... 26	25 ... 21		0
opcode = *COP1*	fmt = *S*		**function**

function bits 5..3		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L	TRUNC.L	CEIL.L	FLOOR.L	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	*MOVCF* δ	MOVZ	MOVN	*	RECIP	RSQRT	
3	011	*	*	*	*	*	*	*	*
4	100	*	CVT.D	*	*	CVT.W	CVT.L	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F α	C.UN α	C.EQ α	C.UEQ α	C.OLT α	C.ULT α	C.OLE α	C.ULE α
7	111	C.SF α	C.NGLE α	C.SEQ α	C.NGL α	C.LT α	C.NGE α	C.LE α	C.NGT α

(columns: function bits 2..0)

encoding when fmt = *D*

31 ... 26	25 ... 21		0
opcode = *COP1*	fmt = *D*		**function**

function bits 5..3		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L	TRUNC.L	CEIL.L	FLOOR.L	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	*MOVCF* δ	MOVZ	MOVN	*	RECIP	RSQRT	
3	011	*	*	*	*	*	*	*	*
4	100	CVT.S	*	*	*	CVT.W	CVT.L	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F α	C.UN α	C.EQ α	C.UEQ α	C.OLT α	C.ULT α	C.OLE α	C.ULE α
7	111	C.SF α	C.NGLE α	C.SEQ α	C.NGL α	C.LT α	C.NGE α	C.LE α	C.NGT α

(columns: function bits 2..0)

encoding when fmt = *W* or *L*

31–26	25–21		0
opcode = *COP1*	fmt = *W, L*		**function**

function bits 5..3		bits 2..0: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	*	*	*	*	*	*	*	*
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	CVT.S	CVT.D	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Table B-26 (cont.) FPU (CP1) Instruction Encoding - MIPS IV Architecture

Instructions encoded by the function field when opcode=*COP1X*.

31–26		5–0
opcode = *COP1X*		**function**

function bits 5..3		bits 2..0: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	LWXC1	LDXC1	*	*	*	*	*	*
1	001	SWXC1	SDXC1	*	*	*	*	*	PREFX
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	MADD.S	MADD.D	*	*	*	*	*	*
5	101	MSUB.S	MSUB.D	*	*	*	*	*	*
6	110	NMADD.S	NMADD.D	*	*	*	*	*	*
7	111	NMSUB.S	NMSUB.D	*	*	*	*	*	*

Instructions encoded by the tf field when opcode=*COP1*, fmt = *S* or *D*, and function=*MOVCF*.

31–26	25–21		16		5–0
opcode = *COP1*	fmt = *S, D*		**tf**		function = *MOVCF*

tf	bit 16: 0	1
	MOVF (fmt)	MOVT (fmt)

These are the MOVF.fmt and MOVT.fmt instructions. They should not be confused with MOVF and MOVT.

Instruction class encoded by the function field when opcode=*SPECIAL*.

31 26		5 0
opcode = *SPECIAL*		**function**

function bits 5..3	bits 2..0 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000		*MOVCI* δ						
...					χ			
7 111								

Instructions encoded by the tf field when opcode = *SPECIAL* and function=*MOVCI*.

31 26		16		5 0
opcode = *SPECIAL*		**tf**		function = *MOVCI*

tf bit 16	0	1
	MOVF	MOVT

These are the MOVF and MOVT instructions. They should not be confused with MOVF.fmt and MOVT.fmt.

Table B-27 Architecture Level In Which FPU Instructions are Defined or Extended.

The architecture level in which each MIPS IVencoding was defined is indicated by a subscript 1, 2, 3, or 4 (for architecture level I, II, III, or IV). If an instruction or instruction class was later extended, the extending level is indicated after the defining level.

Instructions encoded by the opcode field.

31 26	0
opcode	

Architecture level is shown by a subscript 1, 2, III, or 4.

opcode bits 31..29 \ bits 28..26	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	*SPECIAL* β_4							
1 001								
2 010		*COP1* $_{1,2,3,4}$		*COP1X* $_4$		χ		
3 011								
4 100								
5 101								
6 110		LWC1 $_1$				LDC1 $_2$		
7 111		SWC1 $_1$				SDC1 $_2$		

Instructions encoded by the fmt field when opcode=*COP1*.

31 26	25 21	0
opcode = *COP1*	**fmt**	

Architecture level is shown by a subscript 1, 2, 3, or 4.

fmt bits 25..24 \ bits 23..21	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00	MFC1 $_1$	DMFC1 $_3$	CFC1 $_1$	* $_1$	MTC1 $_1$	DMTC1 $_3$	CTC1 $_1$	* $_1$
1 01	*BC* $_{1,2,4}$	* $_1$	* $_1$	* $_1$	* $_1$	* $_1$	* $_1$	* $_1$
2 10	*S* $_{1,2,3,4}$	*D* $_{1,2,3,4}$	* $_1$	* $_1$	*W* $_{1,2,3,4}$	*L* $_{3,4}$	* $_1$	* $_1$
3 11	* $_1$	* $_1$	* $_1$	* $_1$	* $_1$	* $_1$	* $_1$	* $_1$

Instructions encoded by the nd and tf fields when opcode=*COP1* and fmt=*BC*.

31 26	25 21		17	16	0
opcode = *COP1*	fmt = *BC*		**nd**	**tf**	

Architecture level is shown by a subscript 1, 2, 3, or 4.

nd bit 17 \ tf bit 16	0	1
0	BC1F $_{1,4}$	BC1T $_{1,4}$
1	BC1FL $_{2,4}$	BC1TL $_{2,4}$

Table B-27 (cont.) Architecture Level (I-IV) In Which FPU Instructions are Defined or Extended

Instructions encoded by the function field when opcode=*COP1* and fmt = *S*, *D*, *W*, or *L*

encoding when fmt = *S*

31–26	25–21		0
opcode = *COP1*	fmt = *S*		**function**

Architecture level is shown by a subscript 1, 2, 3, or 4.

function bits 5..3 \ bits 2..0		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	ADD $_{1}$	SUB $_{1}$	MUL $_{1}$	DIV $_{1}$	SQRT $_{2}$	ABS $_{1}$	MOV $_{1}$	NEG $_{1}$
1	001	ROUND.L $_{3}$	TRUNC.L $_{3}$	CEIL.L $_{3}$	FLOOR.L $_{3}$	ROUND.W $_{2}$	TRUNC.W $_{2}$	CEIL.W $_{2}$	FLOOR.W $_{2}$
2	010	* $_{1}$	*MOVCF* $_{4}$	MOVZ $_{4}$	MOVN $_{4}$	* $_{1}$	RECIP $_{4}$	RSQRT $_{4}$	* $_{1}$
3	011	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$
4	100	* $_{1}$	CVT.D $_{1,3}$	* $_{1}$	* $_{1}$	CVT.W $_{1}$	CVT.L $_{3}$	* $_{1}$	* $_{1}$
5	101	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$
6	110	C.F $_{1,4}$	C.UN $_{1,4}$	C.EQ $_{1,4}$	C.UEQ $_{1,4}$	C.OLT $_{1,4}$	C.ULT $_{1,4}$	C.OLE $_{1,4}$	C.ULE $_{1,4}$
7	111	C.SF $_{1,4}$	C.NGLE $_{1,4}$	C.SEQ $_{1,4}$	C.NGL $_{1,4}$	C.LT $_{1,4}$	C.NGE $_{1,4}$	C.LE $_{1,4}$	C.NGT $_{1,4}$

encoding when fmt = *D*

31–26	25–21		0
opcode = *COP1*	fmt = *D*		**function**

Architecture level is shown by a subscript 1, 2, 3, or 4.

function bits 5..3 \ bits 2..0		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	ADD $_{1}$	SUB $_{1}$	MUL $_{1}$	DIV $_{1}$	SQRT $_{2}$	ABS $_{1}$	MOV $_{1}$	NEG $_{1}$
1	001	ROUND.L $_{3}$	TRUNC.L $_{3}$	CEIL.L $_{3}$	FLOOR.L $_{3}$	ROUND.W $_{2}$	TRUNC.W $_{2}$	CEIL.W $_{2}$	FLOOR.W $_{2}$
2	010	* $_{1}$	*MOVCF* $_{4}$	MOVZ $_{4}$	MOVN $_{4}$	* $_{1}$	RECIP $_{4}$	RSQRT $_{4}$	* $_{1}$
3	011	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$
4	100	CVT.S $_{1,3}$	* $_{1}$	* $_{1}$	* $_{1}$	CVT.W $_{1}$	CVT.L $_{3}$	* $_{1}$	* $_{1}$
5	101	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$	* $_{1}$
6	110	C.F $_{1,4}$	C.UN $_{1,4}$	C.EQ $_{1,4}$	C.UEQ $_{1,4}$	C.OLT $_{1,4}$	C.ULT $_{1,4}$	C.OLE $_{1,4}$	C.ULE $_{1,4}$
7	111	C.SF $_{1,4}$	C.NGLE $_{1,4}$	C.SEQ $_{1,4}$	C.NGL $_{1,4}$	C.LT $_{1,4}$	C.NGE $_{1,4}$	C.LE $_{1,4}$	C.NGT $_{1,4}$

encoding when
fmt = *W* or *L*

31–26	25–21		5–0
opcode = *COP1*	fmt = *W, L*		**function**

Architecture level is shown by a subscript 1, 2, 3, or 4.

function bits 5..3 / bits 2..0		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$
1	001	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$
2	010	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$
3	011	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$
4	100	CVT.S $_{1,3}$	CVT.D $_{1,3}$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$
5	101	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$
6	110	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$
7	111	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$	$*_1$

Table B-27 (cont.) Architecture Level (I-IV) In Which FPU Instructions are Defined or Extended

Instructions encoded by the function field when opcode=*COP1X*.

31–26		5–0
opcode = *COP1X*		**function**

Architecture level is shown by a subscript 1, 2, 3, or 4.

function bits 5..3 / bits 2..0		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	LWXC1 $_4$	LDXC1 $_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$
1	001	SWXC1 $_4$	SDXC1 $_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$	PREFX $_4$
2	010	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$
3	011	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$
4	100	MADD.S $_4$	MADD.D $_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$
5	101	MSUB.S $_4$	MSUB.D $_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$
6	110	NMADD.S $_4$	NMADD.D $_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$
7	111	NMSUB.S $_4$	NMSUB.D $_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$	$*_4$

Instructions encoded by the tf field when opcode=*COP1*, fmt = *S* or *D*, and function=*MOVCF*.

31–26	25–21		16		5–0
opcode = *COP1*	fmt = *S, D*		**tf**		function = *MOVCF*

tf bit 16	0	1
	MOVF (fmt) $_4$	MOVT (fmt) $_4$

These are the MOVF.fmt and MOVT.fmt instructions. They should not be confused with MOVF and MOVT.

Instruction class encoded by the function field when opcode=*SPECIAL*.

31 26		5 0
opcode = *SPECIAL*		**function**

Architecture level is shown by a subscript 1, 2, 3, or 4.

function bits 5..3	bits 2..0	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000		*MOVCI* $_4$						
	...					χ			
7	111								

Instructions encoded by the tf field when opcode = *SPECIAL* and function=*MOVCI*.

31 26		16		5 0
opcode = *SPECIAL*		**tf**		function = *MOVCI*

tf	bit 16	0	1
		MOVF $_4$	MOVT $_4$

These are the MOVF and MOVT instructions. They should not be confused with MOVF.fmt and MOVT.fmt.

Table B-28 FPU Instruction Encoding Changes - MIPS II Architecture Revision.

An instruction encoding is shown if the instruction is added or extended in this architecture revision. An instruction class, like COP1, is shown if the instruction class is added in this architecture revision.

Instructions encoded by the opcode field.

31 ... 26	... 0
opcode	

opcode bits 31..29	bits 28..26: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001								
2 010								
3 011								
4 100								
5 101								
6 110						LDC1		
7 111						SDC1		

Instructions encoded by the fmt field when opcode=*COP1*.

31 ... 26	25 ... 21	... 0
opcode = *COP1*	**fmt**	

fmt bits 25..24	bits 23..21: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00								
1 01								
2 10								
3 11								

Instructions encoded by the nd and tf fields when opcode=*COP1* and fmt=*BC*.

31 ... 26	25 ... 21		17	16	... 0
opcode = *COP1*	fmt = *BC*		**nd**	**tf**	

nd bit 17	tf bit 16: 0	1
0		
1	BC1FL	BC1TL

Table B-28 (cont.) FPU Instruction Encoding Changes - MIPS II Revision.

Instructions encoded by the function field when opcode=*COP1* and fmt = *S*, *D*, or *W*

encoding when fmt = *S*

31–26	25–21		0
opcode = *COP1*	fmt = *S*		**function**

function bits 5..3	bits 2..0: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000					SQRT			
1 001					ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2 010								
3 011								
4 100								
5 101								
6 110								
7 111								

encoding when fmt = *D*

31–26	25–21		0
opcode = *COP1*	fmt = *D*		**function**

function bits 5..3	bits 2..0: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000					SQRT			
1 001					ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2 010								
3 011								
4 100								
5 101								
6 110								
7 111								

encoding when
fmt = *W*

31 … 26	25 … 21		0
opcode = *COP1*	fmt = *W*		**function**

function bits 5..3		bits 2..0 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000								
1	001								
2	010								
3	011								
4	100								
5	101								
6	110								
7	111								

Table B-29 FPU Instruction Encoding Changes - MIPS III Revision.

An instruction encoding is shown if the instruction is added or extended in this architecture revision. An instruction class, like COP1, is shown if the instruction class is added in this architecture revision.

Instructions encoded by the opcode field.

31 26	0
opcode	

opcode bits 31..29		bits 28..26 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000								
1	001								
2	010								
3	011								
4	100								
5	101								
6	110								
7	111								

Instructions encoded by the fmt field when opcode=*COP1*.

31 26	25 21	0
opcode = *COP1*	**fmt**	

fmt bits 25..24		bits 23..21 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	00		DMFC1				DMTC1		
1	01								
2	10						*L* δ		
3	11								

Instructions encoded by the nd and tf fields when opcode=*COP1* and fmt=*BC*.

31 26	25 21		17	16	0
opcode = *COP1*	fmt = *BC*		**nd**	**tf**	

nd bit 17	tf bit 16: 0	1
0		
1	BC1FL	BC1TL

Table B-29 (cont.) FPU Instruction Encoding Changes - MIPS III Revision.

Instructions encoded by the function field when opcode=*COP1* and fmt = *S*, *D*, or *L*.

encoding when fmt = *S*

31–26	25–21		0
opcode = *COP1*	fmt = *S*		**function**

function bits 5..3 \ bits 2..0	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001	ROUND.L	TRUNC.L	CEIL.L	FLOOR.L				
2 010								
3 011								
4 100						CVT.L		
5 101								
6 110								
7 111								

encoding when fmt = *D*

31–26	25–21		0
opcode = *COP1*	fmt = *D*		**function**

function bits 5..3 \ bits 2..0	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001	ROUND.L	TRUNC.L	CEIL.L	FLOOR.L				
2 010								
3 011								
4 100						CVT.L		
5 101								
6 110								
7 111								

encoding when fmt = *L*

31–26	25–21		0
opcode = *COP1*	fmt = *L*		**function**

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

Table B-30 FPU Instruction Encoding Changes - MIPS IV Revision.

An instruction encoding is shown if the instruction is added or extended in this architecture revision. An instruction class, like COP1X, is shown if the instruction class is added in this architecture revision.

Instructions encoded by the opcode field.

31 26	0
opcode	

opcode bits 31..29	bits 28..26 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001								
2 010				*COP1X* δ				
3 011								
4 100								
5 101								
6 110								
7 111								

Instructions encoded by the fmt field when opcode=*COP1*.

31 26	25 21	0
opcode = *COP1*	**fmt**	

fmt bits 25..24	bits 23..21 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00								
1 01								
2 10								
3 11								

Instructions encoded by the nd and tf fields when opcode=*COP1* and fmt=*BC*.

31 26	25 21		17	16	0
opcode = *COP1*	fmt = *BC*		**nd**	**tf**	

nd bit 17	tf bit 16 0	1
0	BC1F	BC1T
1	BC1FL	BC1TL

Table B-30 (cont.) FPU Instruction Encoding Changes - MIPS IV Revision.

Instructions encoded by the function field when opcode=*COP1* and fmt = *S*, *D*, *W*, or *L*.

encoding when fmt = *S*

31–26	25–21		5–0
opcode = *COP1*	fmt = *S*		**function**

function bits 5..3	bits 2..0 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001								
2 010		*MOVCF* δ	MOVZ	MOVN		RECIP	RSQRT	
3 011								
4 100								
5 101								
6 110	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7 111	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

encoding when fmt = *D*

31–26	25–21		5–0
opcode = *COP1*	fmt = *D*		**function**

function bits 5..3	bits 2..0 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001								
2 010		*MOVCF* δ	MOVZ	MOVN		RECIP	RSQRT	
3 011								
4 100								
5 101								
6 110	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7 111	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

encoding when fmt = *W* or *L*

31–26	25–21		5–0
opcode = *COP1*	fmt = *W*, *L*		**function**

function bits 5..3	bits 2..0 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000								
1 001								
2 010								
3 011								
4 100								
5 101								
6 110								
7 111								

Table B-30 (cont.) FPU Instruction Encoding Changes - MIPS IV Revision.

Instructions encoded by the function field when opcode=*COP1X*.

31 26		5 0
opcode = *COP1X*		**function**

function bits 5..3		bits 2..0: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	LWXC1	LDXC1	*	*	*	*	*	*
1	001	SWXC1	SDXC1	*	*	*	*	*	PREFX
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	MADD.S	MADD.D	*	*	*	*	*	*
5	101	MSUB.S	MSUB.D	*	*	*	*	*	*
6	110	NMADD.S	NMADD.D	*	*	*	*	*	*
7	111	NMSUB.S	NMSUB.D	*	*	*	*	*	*

Instructions encoded by the tf field when opcode=*COP1*, fmt = *S* or *D*, and function=*MOVCF*.

31 26	25 21		16		5 0
opcode = *COP1*	fmt = *S, D*		**tf**		function = *MOVCF*

tf bit 16	0	1
	MOVF (fmt)	MOVT (fmt)

These are the MOVF.fmt and MOVT.fmt instructions. They should not be confused with MOVF and MOVT.

Instruction class encoded by the function field when opcode=*SPECIAL*.

31 26		5 0
opcode = *SPECIAL*		**function**

function bits 5..3		bits 2..0: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000		*MOVCI* δ						
	...					χ			
7	111								

Instructions encoded by the tf field when opcode = *SPECIAL* and function=*MOVCI*.

31 26		16		5 0
opcode = *SPECIAL*		tf		function = *MOVCI*

tf bit 16	0	1
	MOVF	MOVT

These are the MOVF and MOVT instructions. They should not be confused with MOVF.fmt and MOVT.fmt.

Key to all FPU (CP1) instruction encoding tables:

- * This opcode is reserved for future use. An attempt to execute it causes either a Reserved Instruction exception or a Floating Point Unimplemented Operation Exception. The choice of exception is implementation specific.
- α The table shows 16 compare instructions with values named *C*.condition where "condition" is a comparison condition such as "EQ". These encoding values are all documented in the instruction description titled "C.cond.fmt".
- β The SPECIAL instruction class was defined in MIPS I for CPU instructions. An FPU instruction was first added to the instruction class in MIPS IV.
- δ (also *italic* opcode name) This opcode indicates an instruction class. The instruction word must be further decoded by examing additional tables that show values for another instruction field.
- λ The *COP1X* opcode in MIPS IV was the COP3 opcode in MIPS I and II and a reserved instruction in MIPS III.
- χ These opcodes are not FPU operations. For further information on them, look in the CPU Instruction Encoding information section A 8.
- (fmt) This opcode is a conditional move of formatted FP registers - either MOVF.D, MOVF.S, MOVT.D, or MOVT.S. It should not be confused with the similarly-named MOVF or MOVT instruction that moves CPU registers.