# MIPS

## T E C H N O L O G I E S

# MIPS64™ Architecture For Programmers
# Volume II: The MIPS64™ Instruction Set

# Table of Contents

# List of Figures

# List of Tables

# About This Book

The MIPS64™ Architecture For Programmers Volume II comes as a multi-volume set.

- Volume I describes conventions used throughout the document set, and provides an introduction to the MIPS64™ Architecture

- Volume II provides detailed descriptions of each instruction in the MIPS64™ instruction set

- Volume III describes the MIPS64™ Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS64™ processor implementation

- Volume IV-a describes the MIPS16™ Application-Specific Extension to the MIPS64™ Architecture

- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64™ Architecture

- Volume IV-c describes the MIPS-3D™ Application-Specific Extension to the MIPS64™ Architecture

- Volume IV-d describes the SmartMIPS™Application-Specific Extension to the MIPS32™ Architecture and is not applicable to the MIPS64™ document set

## 1.1 Typographical Conventions

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

### 1.1.1 Italic Text

- is used for *emphasis*

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

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

### 1.1.2 Bold Text

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

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

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

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

### 1.1.3 Courier Text

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

## 1.2 UNPREDICTABLE and UNDEFINED

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

### 1.2.1 UNPREDICTABLE

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

**UNPREDICTABLE** results or operations have several implementation restrictions:

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

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

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

### 1.2.2 UNDEFINED

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

**UNDEFINED** operations or behavior has one implementation restriction:

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

## 1.3 Special Symbols in Pseudocode Notation

In this book, algorithmic descriptions of an operation are described as pseudocode in a high-level language notation resembling Pascal. Special symbols used in the pseudocode notation are listed in Table 1-1.

**Table 1-1 Symbols Used in Instruction Operation Statements**

| Symbol | Meaning |
|---|---|
| ← | Assignment |
| =, ≠ | Tests for equality and inequality |
| ‖ | Bit string concatenation |
| $x^y$ | A $y$-bit string formed by $y$ copies of the single-bit value $x$ |
| b#n | A constant value $n$ in base $b$. For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10. |
| $x_{y..z}$ | Selection of bits $y$ through $z$ of bit string $x$. Little-endian bit notation (rightmost bit is 0) is used. If $y$ is less than $z$, this expression is an empty (zero length) bit string. |
| +, − | 2's complement or floating point arithmetic: addition, subtraction |
| ∗, × | 2's complement or floating point multiplication (both used for either) |
| div | 2's complement integer division |
| mod | 2's complement modulo |
| / | Floating point division |
| < | 2's complement less-than comparison |
| > | 2's complement greater-than comparison |
| ≤ | 2's complement less-than or equal comparison |
| ≥ | 2's complement greater-than or equal comparison |
| nor | Bitwise logical NOR |
| xor | Bitwise logical XOR |
| and | Bitwise logical AND |
| or | Bitwise logical OR |
| GPRLEN | The length in bits (32 or 64) of the CPU general-purpose registers |
| *GPR[x]* | CPU general-purpose register $x$. The content of *GPR[0]* is always zero. |
| *FPR[x]* | Floating Point operand register $x$ |
| *FCC[CC]* | Floating Point condition code CC. *FCC[0]* has the same value as *COC[1]*. |
| *FPR[x]* | Floating Point (Coprocessor unit 1), general register $x$ |
| *CPR[z,x,s]* | Coprocessor unit $z$, general register $x$, select $s$ |
| *CCR[z,x]* | Coprocessor unit $z$, control register $x$ |
| *COC[z]* | Coprocessor unit $z$ condition signal |
| *Xlat[x]* | Translation of the MIPS16 GPR number $x$ into the corresponding 32-bit GPR number |
| BigEndianMem | Endian mode as configured at chip reset (0 →Little-Endian, 1 → Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution. |

**Table 1-1 Symbols Used in Instruction Operation Statements**

| Symbol | Meaning |
|---|---|
| BigEndianCPU | The endianness for load and store instructions ($0 \rightarrow$ Little-Endian, $1 \rightarrow$ Big-Endian). In User mode, this endianness may be switched by setting the *RE* bit in the *Status* register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian). |
| ReverseEndian | Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as ($SR_{RE}$ and User mode). |
| *LLbit* | Bit of **virtual** state used to specify operation for instructions that provide atomic read-modify-write. *LLbit* is set when a linked load occurs; it is tested and cleared by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions. |
| **I**:, **I+n**:, **I-n**: | This occurs as a prefix to *Operation* description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to "execute." Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of **I**. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction **I**, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled **I+1**.<br><br>The effect of pseudocode statements for the current instruction labelled **I+1** appears to occur "at the same time" as the effect of pseudocode statements labeled **I** for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur "at the same time," there is no defined order. Programs must not depend on a particular order of evaluation between such sections. |
| PC | The *Program Counter* value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to *PC* during an instruction time. If no value is assigned to *PC* during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16 instruction) or 4 before the next instruction time. A taken branch assigns the target address to the *PC* during the instruction time of the instruction in the branch delay slot. |
| PABITS | The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes. |
| SEGBITS | The number of virtual address bits implemented in a segment of the address space is represented by the symbol SEGBITS. As such, if 40 virtual address bits are implemented in a segment, the size of the segment is $2^{SEGBITS} = 2^{40}$ bytes. |
| FP32RegistersMode | Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR.<br><br>In MIPS32 implementations, **FP32RegistersMode** is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case **FP32RegisterMode** is computed from the FR bit in the *Status* register. If this bit is a 0, the processor operates as if it had 32 32-bit FPRs. If this bit is a 1, the processor operates with 32 64-bit FPRs.<br><br>The value of **FP32RegistersMode** is computed from the FR bit in the *Status* register. |
| InstructionInBranchDelaySlot | Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the *dynamic* state of the instruction, not the *static* state. That is, the value is false if a branch or jump occurs to an instruction whose PC immediately follows a branch or jump, but which is not executed in the delay slot of a branch or jump. |
| SignalException(exception, argument) | Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument). Control does not return from this pseudocode function - the exception is signaled at the point of the call. |

## 1.4  For More Information

Various MIPS RISC processor manuals and additional information about MIPS products can be found at the MIPS URL:

http://www.mips.com

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

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

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

# Guide to the Instruction Set

This chapter provides a detailed guide to understanding the instruction descriptions, which are listed in alphabetical order in the tables at the beginning of the next chapter.

## 2.1  Understanding the Instruction Fields

Figure 2-1 shows an example instruction. Following the figure are descriptions of the fields listed below:

- "Instruction Fields" on page 8
- "Instruction Descriptive Name and Mnemonic" on page 9
- "Format Field" on page 9
- "Purpose Field" on page 10
- "Description Field" on page 10
- "Restrictions Field" on page 10
- "Operation Field" on page 11
- "Exceptions Field" on page 11
- "Programming Notes and Implementation Notes Fields" on page 11

Instruction Mnemonic
and Descriptive Name

**Example Instruction Name**                                                    **EXAMPLE**

Instruction encoding
constant and variable
field names and values

| 31        26 | 25      21 | 20      16 | 15      11 | 10      6 | 5      0 |
|---|---|---|---|---|---|
| SPECIAL<br>0 0 0 0 0 0 | rs | rt | rd | 0<br>0 0 0 0 0 | EXAMPLE<br>0 0 0 0 0 0 |
| 6 | 5 | 5 | 5 | 5 | 6 |

Architecture level at
which instruction was
defined/redefined and
assembler format(s) for
each definition

**Format:** EXAMPLE rd, rs,rt                                                    **MIPS32**

Short description

**Purpose:** to execute an EXAMPLE op

Symbolic description

Full description of
instruction operation

**Description:** rd ← rs exampleop rt
This section describes the operation of the instruction in text, tables, and
illustrations. It includes information that would be difficult to encode in the
Operation section.

Restrictions on
instruction and
operands

**Restrictions:**
This section lists any restrictions for the instruction. This can include values of the
instruction encoding fields such as register specifiers, operand values, operand
formats, address alignment, instruction scheduling hazards, and type of memory
access for addressed locations.

High-level language
description of instruction
operation

**Operation:**
/* This section describes the operation of an instruction in a */
/* high-level pseudo-language. It is precise in ways that the */
/* Description section is not, but is also missing information */
/* that is hard to express in pseudocode.*/
     temp    ← GPR[rs] exampleop GPR[rt]
     GPR[rd]← sign_extend(temp$_{31..0}$)

Exceptions that
instruction can cause

**Exceptions:**
A list of exceptions taken by the instruction

Notes for programmers

**Programming Notes:**
Information useful to programmers, but not necessary to describe the operation of
the instruction

Notes for implementors

**Implementation Notes:**
Like *Programming Notes*, except for processor implementors

**Figure 2-1 Example of Instruction Description**

### 2.1.1 Instruction Fields

Fields encoding the instruction word are shown in register form at the top of the instruction description. The following
rules are followed:

- The values of constant fields and the *opcode* names are listed in uppercase (SPECIAL and ADD in Figure 2-2). Constant values in a field are shown in binary below the symbolic or hexadecimal value.

- All variable fields are listed with the lowercase names used in the instruction description (*rs*, *rt* and *rd* in Figure 2-2).

- Fields that contain zeros but are not named are unused fields that are required to be zero (bits 10:6 in Figure 2-2). If such fields are set to non-zero values, the operation of the processor is **UNPREDICTABLE**.

| 31        26 | 25       21 | 20       16 | 15       11 | 10        6 | 5         0 |
|:---:|:---:|:---:|:---:|:---:|:---:|
| SPECIAL<br>000000 | rs | rt | rd | 0<br>00000 | ADD<br>100000 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Figure 2-2 Example of Instruction Fields**

### 2.1.2 Instruction Descriptive Name and Mnemonic

The instruction descriptive name and mnemonic are printed as page headings for each instruction, as shown in Figure 2-3.

| | |
|:---|---:|
| **Add Word** | **ADD** |

**Figure 2-3 Example of Instruction Descriptive Name and Mnemonic**

### 2.1.3 Format Field

The assembler formats for the instruction and the architecture level at which the instruction was originally defined are given in the *Format* field. If the instruction definition was later extended, the architecture levels at which it was extended and the assembler formats for the extended definition are shown in their order of extension (for an example, see C.cond.fmt). The MIPS architecture levels are inclusive; higher architecture levels include all instructions in previous levels. Extensions to instructions are backwards compatible. The original assembler formats are valid for the extended architecture.

**Format:**  `ADD rd, rs, rt`                                                    **MIPS32 (MIPS I)**

**Figure 2-4 Example of Instruction Format**

The assembler format is shown with literal parts of the assembler instruction printed in uppercase characters. The variable parts, the operands, are shown as the lowercase names of the appropriate fields. The architectural level at which the instruction was first defined, for example "MIPS32" is shown at the right side of the page. If the instruction was originally defined in the MIPS I through MIPS V levels of the architecture, that information is enclosed in parentheses.

There can be more than one assembler format for each architecture level. Floating point operations on formatted data show an assembly format with the actual assembler mnemonic for each valid value of the *fmt* field. For example, the ADD.fmt instruction lists both ADD.S and ADD.D.

The assembler format lines sometimes include parenthetical comments to help explain variations in the formats (once again, see C.cond.fmt). These comments are not a part of the assembler format.

### 2.1.4 Purpose Field

The *Purpose* field gives a short description of the use of the instruction.

---

**Purpose:**

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

**Figure 2-5 Example of Instruction Purpose**

---

### 2.1.5 Description Field

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

---

**Description:** 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, the destination register is not modified and an Integer Overflow exception occurs

- If the addition does not overflow, the 32-bit result is signed-extended and placed into GPR *rd*

**Figure 2-6 Example of Instruction Description**

---

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

This section uses acronyms for register descriptions. "GPR *rt*" is CPU general-purpose register specified by the instruction field *rt*. "FPR *fs*" is the floating point operand register specified by the instruction field *fs*. "CP1 register *fd*" is the coprocessor 1 general register specified by the instruction field *fd*. "*FCSR*" is the floating point *Control /Status* register.

### 2.1.6 Restrictions Field

The *Restrictions* field documents any possible restrictions that may affect the instruction. Most restrictions fall into one of the following six categories:

- Valid values for instruction fields (for example, see floating point ADD.fmt)

- ALIGNMENT requirements for memory addresses (for example, see LW)

- Valid values of operands (for example, see DADD)

- Valid operand formats (for example, see floating point ADD.fmt)

- Order of instructions necessary to guarantee correct execution. These ordering constraints avoid pipeline hazards for which some processors do not have hardware interlocks (for example, see MUL).

- Valid memory access types (for example, see LL/SC)

**Restrictions:**

If either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

**Figure 2-7 Example of Instruction Restrictions**

### 2.1.7 Operation Field

The *Operation* field describes the operation of the instruction as pseudocode in a high-level language notation resembling Pascal. This formal description complements the *Description* section; it is not complete in itself because many of the restrictions are either difficult to include in the pseudocode or are omitted for legibility.

**Operation:**

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← (GPR[rs]₃₁||GPR[rs]₃₁..₀) + (GPR[rt]₃₁||GPR[rt]₃₁..₀)
if temp₃₂ ≠ temp₃₁ then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← sign_extend(temp₃₁..₀)
endif
```

**Figure 2-8 Example of Instruction Operation**

See Section 2.2 , "Operation Section Notation and Functions" on page 12 for more information on the formal notation used here.

### 2.1.8 Exceptions Field

The *Exceptions* field lists the exceptions that can be caused by *Operation* of the instruction. It omits exceptions that can be caused by the instruction fetch, for instance, TLB Refill, and also omits exceptions that can be caused by asynchronous external events such as an Interrupt. Although a Bus Error exception may be caused by the operation of a load or store instruction, this section does not list Bus Error for load and store instructions because the relationship between load and store instructions and external error indications, like Bus Error, are dependent upon the implementation.

**Exceptions:**

Integer Overflow

**Figure 2-9 Example of Instruction Exception**

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

### 2.1.9 Programming Notes and Implementation Notes Fields

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

**Programming Notes:**

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

**Figure 2-10 Example of Instruction Programming Notes**

## 2.2 Operation Section Notation and Functions

In an instruction description, the *Operation* section uses a high-level language notation to describe the operation performed by each instruction. Special symbols used in the pseudocode are described in the previous chapter. Specific pseudocode functions are described below.

This section presents information about the following topics:

- "Instruction Execution Ordering" on page 12
- "Pseudocode Functions" on page 12

### 2.2.1 Instruction Execution Ordering

Each of the high-level language statements in the *Operations* section are executed sequentially (except as constrained by conditional and loop constructs).

### 2.2.2 Pseudocode Functions

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

- "Coprocessor General Register Access Functions" on page 12
- "Load Memory and Store Memory Functions" on page 14
- "Access Functions for Floating Point Registers" on page 16
- "Miscellaneous Functions" on page 18

#### 2.2.2.1 Coprocessor General Register Access Functions

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

#### COP_LW

The COP_LW function defines the action taken by coprocessor z when supplied with a word from memory during a load word operation. The action is coprocessor-specific. The typical action would be to store the contents of memword in coprocessor general register *rt*.

```
COP_LW (z, rt, memword)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    memword: A 32-bit word value supplied to the coprocessor

    /* Coprocessor-dependent action */

endfunction COP_LW
```

**Figure 2-11 COP_LW Pseudocode Function**

### COP_LD

The COP_LD function defines the action taken by coprocessor z when supplied with a doubleword from memory during a load doubleword operation. The action is coprocessor-specific. The typical action would be to store the contents of memdouble in coprocessor general register *rt*.

```
COP_LD (z, rt, memdouble)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    memdouble:  64-bit doubleword value supplied to the coprocessor.

    /* Coprocessor-dependent action */

endfunction COP_LD
```

**Figure 2-12 COP_LD Pseudocode Function**

### COP_SW

The COP_SW function defines the action taken by coprocessor z to supply a word of data during a store word operation. The action is coprocessor-specific. The typical action would be to supply the contents of the low-order word in coprocessor general register *rt*.

```
dataword ← COP_SW (z, rt)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    dataword: 32-bit word value

    /* Coprocessor-dependent action */

endfunction COP_SW
```

**Figure 2-13 COP_SW Pseudocode Function**

### COP_SD

The COP_SD function defines the action taken by coprocessor z to supply a doubleword of data during a store doubleword operation. The action is coprocessor-specific. The typical action would be to supply the contents of the low-order doubleword in coprocessor general register *rt*.

```
datadouble ← COP_SD (z, rt)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    datadouble: 64-bit doubleword value

    /* Coprocessor-dependent action */

endfunction COP_SD
```

**Figure 2-14 COP_SD Pseudocode Function**

### 2.2.2.2 Load Memory 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 of the bytes that form the object. For big-endian ordering this is the most-significant byte; for a little-endian ordering this is the least-significant byte.

In the *Operation* pseudocode for load and store operations, the following functions summarize the handling of virtual addresses and the access of physical memory. The size of the data item to be loaded or stored is passed in the *AccessLength* field. The valid constant names and values are shown in Table 2-1. The bytes within the addressed unit of memory (word for 32-bit processors or doubleword for 64-bit processors) that are used can be determined directly from the *AccessLength* and the two or three low-order bits of the address.

*AddressTranslation*

The AddressTranslation function translates a virtual address to a physical address and its cache coherence algorithm, describing the mechanism used to resolve the memory reference.

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

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

    /* pAddr: physical address */
    /* CCA:   Cache Coherence Algorithm, the method used to access caches*/
    /*        and memory and resolve the reference */

    /* vAddr: virtual address */
    /* IorD:  Indicates whether access is for INSTRUCTION or DATA */
    /* LorS:  Indicates whether access is for LOAD or STORE */

    /* See the address translation description for the appropriate MMU */
    /* type in Volume III of this book for the exact translation mechanism */

endfunction AddressTranslation
```

**Figure 2-15 AddressTranslation Pseudocode Function**

*LoadMemory*

The LoadMemory function loads a value from memory.

This action uses cache and main memory as specified in both the Cache Coherence Algorithm (*CCA*) and the access (*IorD*) to find the contents of *AccessLength* memory bytes, starting at physical location *pAddr*. The data is returned in a fixed-width naturally aligned memory element (*MemElem*). The low-order 2 (or 3) bits of the address and the *AccessLength* indicate which of the bytes within *MemElem* need to be passed to the processor. If the memory access type of the reference is *uncached*, only the referenced bytes are read from memory and marked as valid within the memory element. If the access type is *cached* but the data is not present in cache, an implementation-specific *size* and *alignment* block of memory is read and loaded into the cache to satisfy a load reference. At a minimum, this block is the entire memory element.

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

    /* MemElem:  Data is returned in a fixed width with a natural alignment. The */
    /*           width is the same size as the CPU general-purpose register, */
    /*           32 or 64 bits, aligned on a 32- or 64-bit boundary, */
    /*           respectively. */
    /* CCA:      Cache Coherence Algorithm, the method used to access caches */
    /*           and memory and resolve the reference */

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

endfunction LoadMemory
```

**Figure 2-16 LoadMemory Pseudocode Function**

### *StoreMemory*

The StoreMemory function stores a value to memory.

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

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

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

endfunction StoreMemory
```

**Figure 2-17 StoreMemory Pseudocode Function**

### *Prefetch*

The Prefetch function prefetches data from memory.

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

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

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

endfunction Prefetch
```

**Figure 2-18 Prefetch Pseudocode Function**

Table 2-1 lists the data access lengths and their labels for loads and stores.

**Table 2-1 AccessLength Specifications for Loads/Stores**

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

### 2.2.2.3 Access Functions for Floating Point Registers

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

***ValueFPR***

The ValueFPR function returns a formatted value from the floating point registers.

```
value ← ValueFPR(fpr, fmt)

    /* value: The formatted value from the FPR */

    /* fpr:   The FPR number */
    /* fmt:   The format of the data, one of: */
    /*        S, D, W, L, PS, */
    /*        OB, QH, */
    /*        UNINTERPRETED_WORD, */
    /*        UNINTERPRETED_DOUBLEWORD */
    /* The UNINTERPRETED values are used to indicate that the datatype */
    /* is not known as, for example, in SWC1 and SDC1 */

    case fmt of
        S, W, UNINTERPRETED_WORD:
            valueFPR ← UNPREDICTABLE^32 ‖ FPR[fpr]_{31..0}

        D, UNINTERPRETED_DOUBLEWORD:
            if (FP32RegistersMode = 0)
                if (fpr_0 ≠ 0) then
                    valueFPR ← UNPREDICTABLE
                else
                    valueFPR ← FPR[fpr+1]_{31..0} ‖ FPR[fpr]_{31..0}
                endif
            else
                valueFPR ← FPR[fpr]
            endif

        L, PS, OB, QH:
            if (FP32RegistersMode = 0) then
                valueFPR ← UNPREDICTABLE
            else
                valueFPR ← FPR[fpr]
            endif

        DEFAULT:
            valueFPR ← UNPREDICTABLE

    endcase
endfunction ValueFPR
```

**Figure 2-19 ValueFPR Pseudocode Function**

### *StoreFPR*

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

```
StoreFPR (fpr, fmt, value)

    /* fpr:   The FPR number */
    /* fmt:   The format of the data, one of: */
    /*        S, D, W, L, PS, */
    /*        OB, QH, */
    /*        UNINTERPRETED_WORD, */
    /*        UNINTERPRETED_DOUBLEWORD */
    /* value: The formattted value to be stored into the FPR */

    /* The UNINTERPRETED values are used to indicate that the datatype */
    /* is not known as, for example, in LWC1 and LDC1 */


    case fmt of
        S, W, UNINTERPRETED_WORD:
```
$$FPR[fpr] \leftarrow \textbf{UNPREDICTABLE}^{32} \parallel value_{31..0}$$

```
        D, UNINTERPRETED_DOUBLEWORD:
            if (FP32RegistersMode = 0)
```
$$\text{if } (fpr_0 \neq 0) \text{ then}$$
```
                    UNPREDICTABLE
                else
```
$$FPR[fpr] \quad\; \leftarrow \textbf{UNPREDICTABLE}^{32} \parallel value_{31..0}$$
$$FPR[fpr+1] \leftarrow \textbf{UNPREDICTABLE}^{32} \parallel value_{63..32}$$
```
                endif
            else
                FPR[fpr] ← value
            endif

        L, PS, OB, QH:
            if (FP32RegistersMode = 0) then
                UNPREDICTABLE
            else
                FPR[fpr] ← value
            endif

    endcase

endfunction StoreFPR
```

**Figure 2-20 StoreFPR Pseudocode Function**

### 2.2.2.4  Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

*SyncOperation*

The SyncOperation function orders loads and stores to synchronize shared memory.

This action makes the effects of the synchronizable loads and stores indicated by *stype* occur in the same order for all processors.

```
SyncOperation(stype)

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

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

endfunction SyncOperation
```

**Figure 2-21 SyncOperation Pseudocode Function**

*SignalException*

The SignalException function signals an exception condition.

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

```
SignalException(Exception, argument)

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

endfunction SignalException
```

**Figure 2-22 SignalException Pseudocode Function**

*NullifyCurrentInstruction*

The NullifyCurrentInstruction function nullifies the current instruction.

The instruction is aborted. For branch-likely instructions, nullification kills the instruction in the delay slot during its execution.

```
NullifyCurrentInstruction()

endfunction NullifyCurrentInstruction
```

**Figure 2-23 NullifyCurrentInstruction PseudoCode Function**

*CoprocessorOperation*

The CoprocessorOperation function performs the specified Coprocessor operation.

```
CoprocessorOperation (z, cop_fun)

    /* z:        Coprocessor unit number */
    /* cop_fun:  Coprocessor function from function field of instruction */

    /* Transmit the cop_fun value to coprocessor z */

endfunction CoprocessorOperation
```

**Figure 2-24 CoprocessorOperation Pseudocode Function**

*JumpDelaySlot*

The JumpDelaySlot function is used in the pseudocode for the four PC-relative instructions. The function returns TRUE if the instruction at *vAddr* is executed in a jump delay slot. A jump delay slot always immediately follows a JR, JAL, JALR, or JALX instruction.

```
JumpDelaySlot(vAddr)

    /* vAddr:Virtual address */

endfunction JumpDelaySlot
```

**Figure 2-25 JumpDelaySlot Pseudocode Function**

*NotWordValue*

The NotWordValue function returns a boolean value that determines whether the 64-bit value contains a valid word (32-bit) value. Such a value has bits 63..32 equal to bit 31.

```
result ← NotWordValue(value)

    /* result:    True if the value is not a correct sign-extended word value; */
    /*            False otherwise */

    /* value:     A 64-bit register value to be checked */
```

$$\text{NotWordValue} \leftarrow \text{value}_{63..32} \neq (\text{value}_{31})^{32}$$

```
endfunction NotWordValue
```

**Figure 2-26 NotWordValue Pseudocode Function**

*FPConditionCode*

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

```
tf ←FPConditionCode(cc)

    /* tf: The value of the specified condition code */

    /* cc: The Condition code number in the range 0..7 */

    if cc = 0 then
        FPConditionCode ← FCSR₂₃
    else
        FPConditionCode ← FCSR₂₄₊cc
    endif

endfunction FPConditionCode
```

**Figure 2-27 FPConditionCode Pseudocode Function**

*SetFPConditionCode*

The SetFPConditionCode function writes a new value to a specific floating point condition code.

```
SetFPConditionCode(cc)
    if cc = 0 then
        FCSR ← FCSR31..24 || tf || FCSR22..0
    else
        FCSR ← FCSR31..25+cc || tf || FCSR23+cc..0
endif

endfunction SetFPConditionCode
```

**Figure 2-28 SetFPConditionCode Pseudocode Function**

## 2.3 Op and Function Subfield Notation

In some instructions, the instruction subfields *op* and *function* can have constant 5- or 6-bit values. When reference is made to these instructions, uppercase mnemonics are used. For instance, in the floating point ADD instruction, *op*=COP1 and *function*=ADD. In other cases, a single field has both fixed and variable subfields, so the name contains both upper- and lowercase characters.

## 2.4 FPU Instructions

In the detailed description of each FPU instruction, all variable subfields in an instruction format (such as *fs, ft, immediate*, and so on) are shown in lowercase. The instruction name (such as ADD, SUB, and so on) is shown in uppercase.

For the sake of clarity, an alias is sometimes used for a variable subfield in the formats of specific instructions. For example, *rs=base* in the format for load and store instructions. Such an alias is always lowercase since it refers to a variable subfield.

Bit encodings for mnemonics are given in Volume I, in the chapters describing the CPU, FPU, MDMX, and MIPS16 instructions.

See Section 2.3 , "Op and Function Subfield Notation" on page 21 for a description of the *op* and *function* subfields.

# The MIPS64™ Instruction Set

## 3.1  Compliance and Subsetting

To be compliant with the MIPS64 Architecture, designs must implement a set of required features, as described in this document set. To allow flexibility in implementations, the MIPS64 Architecture does provide subsetting rules. An implementation that follows these rules is compliant with the MIPS64 Architecture as long as it adheres strictly to the rules, and fully implements the remaining instructions.

The instruction set subsetting rules are as follows:

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

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

  – No FPU

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

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

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

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

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

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

Supersetting of the MIPS64 ISA is only allowed by adding functions to the *SPECIAL2* major opcode or by adding instructions to support Coprocessor 2.

## 3.2  Alphabetical List of Instructions

Table 3-1 through Table 3-23 provide a list of instructions grouped by category. Individual instruction descriptions follow the tables, arranged in alphabetical order.

**Table 3-1 CPU Arithmetic Instructions**

| Mnemonic | Instruction |
| --- | --- |
| ADD | Add Word |
| ADDI | Add Immediate Word |
| ADDIU | Add Immediate Unsigned Word |
| ADDU | Add Unsigned Word |
| CLO | Count Leading Ones in Word |
| CLZ | Count Leading Zeros in Word |
| DADD | Doubleword Add |
| DADDI | Doubleword Add immediate |
| DADDIU | Doubleword Add Immediate Unsigned |
| DADDU | Doubleword Add Unsigned |
| DCLO | Count Leading Ones in Doubleword |
| DCLZ | Count Leading Zeros in Doubleword |
| DDIV | Doubleword Divide |
| DDIVU | Doubleword Divide Unsigned |
| DIV | Divide Word |
| DIVU | Divide Unsigned Word |
| DMULT | Doubleword Multiply |
| DMULTU | Doubleword Multiply Unsigned |
| DSUB | Doubleword Subtract |
| DSUBU | Doubleword Subtract Unsigned |
| MADD | Multiply and Add Word to Hi, Lo |
| MADDU | Multiply and Add Unsigned Word to Hi, Lo |
| MSUB | Multiply and Subtract Word to Hi, Lo |
| MSUBU | Multiply and Subtract Unsigned Word to Hi, Lo |
| MUL | Multiply Word to GPR |
| MULT | Multiply Word |
| MULTU | Multiply Unsigned Word |
| SLT | Set on Less Than |

**Table 3-1 CPU Arithmetic Instructions**

| Mnemonic | Instruction |
|---|---|
| SLTI | Set on Less Than Immediate |
| SLTIU | Set on Less Than Immediate Unsigned |
| SLTU | Set on Less Than Unsigned |
| SUB | Subtract Word |
| SUBU | Subtract Unsigned Word |

**Table 3-2 CPU Branch and Jump Instructions**

| Mnemonic | Instruction |
|---|---|
| B | Unconditional Branch |
| BAL | Branch and Link |
| BEQ | Branch on Equal |
| BGEZ | Branch on Greater Than or Equal to Zero |
| BGEZAL | Branch on Greater Than or Equal to Zero and Link |
| BGTZ | Branch on Greater Than Zero |
| BLEZ | Branch on Less Than or Equal to Zero |
| BLTZ | Branch on Less Than Zero |
| BLTZAL | Branch on Less Than Zero and Link |
| BNE | Branch on Not Equal |
| J | Jump |
| JAL | Jump and Link |
| JALR | Jump and Link Register |
| JR | Jump Register |

**Table 3-3 CPU Instruction Control Instructions**

| Mnemonic | Instruction |
|---|---|
| NOP | No Operation |
| SSNOP | Superscalar No Operation |

**Table 3-4 CPU Load, Store, and Memory Control Instructions**

| Mnemonic | Instruction |
|---|---|
| LB | Load Byte |
| LBU | Load Byte Unsigned |
| LD | Load Doubleword |
| LDL | Load Doubleword LEft |
| LDR | Load Doubleword Right |
| LH | Load Halfword |
| LHU | Load Halfword Unsigned |
| LL | Load Linked Word |
| LLD | Load Linked Doubleword |
| LW | Load Word |
| LWL | Load Word Left |
| LWR | Load Word Right |
| LWU | Load Word Unsigned |
| PREF | Prefetch |
| SB | Store Byte |
| SC | Store Conditional Word |
| SCD | Store Conditional Doubleword |
| SD | Store Doubleword |
| SDL | Store Doubleword LEft |
| SDR | Store Doubleword Right |
| SH | Store Halfword |
| SW | Store Word |
| SWL | Store Word Left |
| SWR | Store Word Right |
| SYNC | Synchronize Shared Memory |

**Table 3-5 CPU Logical Instructions**

| Mnemonic | Instruction |
|---|---|
| AND | And |
| ANDI | And Immediate |
| LUI | Load Upper Immediate |
| NOR | Not Or |
| OR | Or |
| ORI | Or Immediate |
| XOR | Exclusive Or |
| XORI | Exclusive Or Immediate |

**Table 3-6 CPU Move Instructions**

| Mnemonic | Instruction |
|---|---|
| MFHI | Move From HI Register |
| MFLO | Move From LO Register |
| MOVF | Move Conditional on Floating Point False |
| MOVN | Move Conditional on Not Zero |
| MOVT | Move Conditional on Floating Point True |
| MOVZ | Move Conditional on Zero |
| MTHI | Move To HI Register |
| MTLO | Move To LO Register |

**Table 3-7 CPU Shift Instructions**

| Mnemonic | Instruction |
|---|---|
| DSLL | Doubleword Shift Left Logical |
| DSLL32 | Doubleword Shift Left Logical Plus 32 |
| DSLLV | Doubleword Shift Left Logical Variable |
| DSRA | Doubleword Shift Right Arithmetic |
| DSRA32 | Doubleword Shift Right Arithmetic Plus 32 |
| DSRAV | Doubleword Shift Right Arithmetic Variable |
| DSRL | Doubleword Shift Right Logical |
| DSRL32 | Doubleword Shift Right Logical Plus 32 |
| DSRLV | Doubleword Shift Right Logical Variable |
| SLL | Shift Word Left Logical |

**Table 3-7 CPU Shift Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| SLLV | Shift Word Left Logical Variable |
| SRA | Shift Word Right Arithmetic |
| SRAV | Shift Word Right Arithmetic Variable |
| SRL | Shift Word Right Logical |
| SRLV | Shift Word Right Logical Variable |

**Table 3-8 CPU Trap Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| BREAK | Breakpoint |
| SYSCALL | System Call |
| TEQ | Trap if Equal |
| TEQI | Trap if Equal Immediate |
| TGE | Trap if Greater or Equal |
| TGEI | Trap if Greater of Equal Immediate |
| TGEIU | Trap if Greater or Equal Immediate Unsigned |
| TGEU | Trap if Greater or Equal Unsigned |
| TLT | Trap if Less Than |
| TLTI | Trap if Less Than Immediate |
| TLTIU | Trap if Less Than Immediate Unsigned |
| TLTU | Trap if Less Than Unsigned |
| TNE | Trap if Not Equal |
| TNEI | Trap if Not Equal Immediate |

**Table 3-9 Obsolete[a] CPU Branch Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| BEQL | Branch on Equal Likely |
| BGEZALL | Branch on Greater Than or Equal to Zero and Link Likely |
| BGEZL | Branch on Greater Than or Equal to Zero Likely |
| BGTZL | Branch on Greater Than Zero Likely |
| BLEZL | Branch on Less Than or Equal to Zero Likely |
| BLTZALL | Branch on Less Than Zero and Link Likely |
| BLTZL | Branch on Less Than Zero Likely |
| BNEL | Branch on Not Equal Likely |

a. Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS64 architecture.

**Table 3-10 FPU Arithmetic Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| ABS.fmt | Floating Point Absolute Value |
| ADD.fmt | Floating Point Add |
| DIV.fmt | Floating Point Divide |
| MADD.fmt | Floating Point Multiply Add |
| MSUB.fmt | Floating Point Multiply Subtract |
| MUL.fmt | Floating Point Multiply |
| NEG.fmt | Floating Point Negate |
| NMADD.fmt | Floating Point Negative Multiply Add |
| NMSUB.fmt | Floating Point Negative Multiply Subtract |
| RECIP.fmt | Reciprocal Approximation |
| RSQRT.fmt | Reciprocal Square Root Approximation |
| SQRT | Floating Point Square Root |
| SUB.fmt | Floating Point Subtract |

**Table 3-11 FPU Branch Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| BC1F | Branch on FP False |
| BC1T | Branch on FP True |

**Table 3-12 FPU Compare Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| C.cond.fmt | Floating Point Compare |

**Table 3-13 FPU Convert Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| ALNV.PS | Floating Point Align Variable |
| CEIL.L.fmt | Floating Point Ceiling Convert to Long Fixed Point |
| CEIL.W.fmt | Floating Point Ceiling Convert to Word Fixed Point |
| CVT.D.fmt | Floating Point Convert to Double Floating Point |
| CVT.L.fmt | Floating Point Convert to Long Fixed Point |

**Table 3-13 FPU Convert Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| CVT.PS.S | Floating Point Convert Pair to Paired Single |
| CVT.S.PL | Floating Point Convert Pair Lower to Single Floating Point |
| CVT.S.PU | Floating Point Convert Pair Upper to Single Floating Point |
| CVT.S.fmt | Floating Point Convert to Single Floating Point |
| CVT.W.fmt | Floating Point Convert to Word Fixed Point |
| FLOOR.L.fmt | Floating Point Floor Convert to Long Fixed Point |
| FLOOR.W.fmt | Floating Point Floor Convert to Word Fixed Point |
| PLL.PS | Pair Lower Lower |
| PLU.PS | Pair Lower Upper |
| PUL.PS | Pair Upper Lower |
| PUU.PS | Pair Upper Upper |
| ROUND.L.fmt | Floating Point Round to Long Fixed Point |
| ROUND.W.fmt | Floating Point Round to Word Fixed Point |
| TRUNC.L.fmt | Floating Point Truncate to Long Fixed Point |
| TRUNC.W.fmt | Floating Point Truncate to Word Fixed Point |

**Table 3-14 FPU Load, Store, and Memory Control Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| LDC1 | Load Doubleword to Floating Point |
| LDXC1 | Load Doubleword Indexed to Floating Point |
| LUXC1 | Load Doubleword Indexed Unaligned to Floating Point |
| LWC1 | Load Word to Floating Point |
| LWXC1 | Load Word Indexed to Floating Point |
| PREFX | Prefetch Indexed |
| SDC1 | Store Doubleword from Floating Point |
| SDXC1 | Store Doubleword Indexed from Floating Point |
| SUXC1 | Store Doubleword Indexed Unaligned from Floating Point |
| SWC1 | Store Word from Floating Point |
| SWXC1 | Store Word Indexed from Floating Point |

**Table 3-15 FPU Move Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| CFC1 | Move Control Word from Floating Point |
| CTC1 | Move Control Word to Floating Point |
| DMFC1 | Doubleword Move from Floating Point |
| DMTC1 | Doubleword Move to Floating Point |
| MFC1 | Move Word from Floating Point |
| MOV.fmt | Floating Point Move |
| MOVF.fmt | Floating Point Move Conditional on Floating Point False |
| MOVN.fmt | Floating Point Move Conditional on Not Zero |
| MOVT.fmt | Floating Point Move Conditional on Floating Point True |
| MOVZ.fmt | Floating Point Move Conditional on Zero |
| MTC1 | Move Word to Floating Point |

**Table 3-16 Obsolete[a] FPU Branch Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| BC1FL | Branch on FP False Likely |
| BC1TL | Branch on FP True Likely |

a. Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from
    a future revision of the MIPS64 architecture.

**Table 3-17 Coprocessor Branch Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| BC2F | Branch on COP2 False |
| BC2T | Branch on COP2 True |

**Table 3-18 Coprocessor Execute Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| COP2 | Coprocessor Operation to Coprocessor 2 |

**Table 3-19 Coprocessor Load and Store Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| LDC2 | Load Doubleword to Coprocessor 2 |
| LWC2 | Load Word to Coprocessor 2 |
| SDC2 | Store Doubleword from Coprocessor 2 |
| SWC2 | Store Word from Coprocessor 2 |

**Table 3-20 Coprocessor Move Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| CFC2 | Move Control Word from Coprocessor 2 |
| CTC2 | Move Control Word to Coprocessor 2 |
| DMFC2 | Doubleword Move from Coprocessor 2 |
| DMTC2 | Doubleword Move to Coprocessor 2 |
| MFC2 | Move Word from Coprocessor 2 |
| MTC2 | Move Word to Coprocessor 2 |

**Table 3-21 Obsolete[a] Coprocessor Branch Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| BC2FL | Branch on COP2 False Likely |
| BC2TL | Branch on COP2 True Likely |

a. Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS64 architecture.

**Table 3-22 Privileged Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| CACHE | Perform Cache Operation |
| DMFC0 | Doubleword Move from Coprocessor 0 |
| DMTC0 | Doubleword Move to Coprocessor 0 |
| ERET | Exception Return |
| MFC0 | Move from Coprocessor 0 |
| MTC0 | Move to Coprocessor 0 |

**Table 3-22 Privileged Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| TLBP | Probe TLB for Matching Entry |
| TLBR | Read Indexed TLB Entry |
| TLBWI | Write Indexed TLB Entry |
| TLBWR | Write Random TLB Entry |
| WAIT | Enter Standby Mode |

**Table 3-23 EJTAG Instructions**

| Mnemonic | Instruction |
|----------|-------------|
| DERET | Debug Exception Return |
| SDBBP | Software Debug Breakpoint |

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| COP1 010001 | | fmt | | 0 00000 | | fs | | fd | | ABS 000101 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |
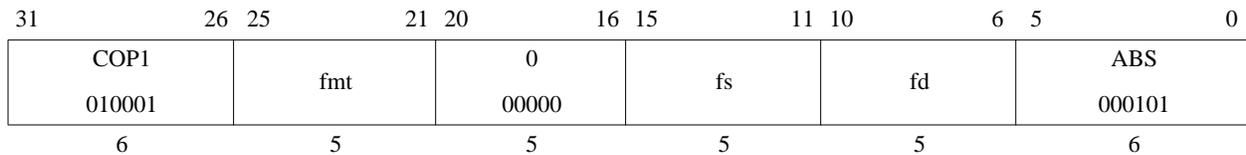
**Format:** ABS.S fd, fs **MIPS32 (MIPS I)**
ABS.D fd, fs **MIPS32 (MIPS I)**
ABS.PS fd, fs **MIPS64 (MIPS V)**

**Purpose:**

To compute the absolute value of an FP value

**Description:** fd ← abs(fs)

The absolute value of the value in FPR *fs* is placed in FPR *fd*. The operand and result are values in format *fmt*. ABS.PS takes the absolute value of the two values in FPR *fs* independently, and ORs together any generated exceptions.

*Cause* bits are ORed into the *Flag* bits if no exception is taken.

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*. If they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of ABS.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
StoreFPR(fd, fmt, AbsoluteValue(ValueFPR(fs, fmt)))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation

## Add Word ADD

| 31        26 | 25        21 | 20        16 | 15        11 | 10        6 | 5        0 |
|---|---|---|---|---|---|
| SPECIAL<br>000000 | rs | rt | rd | 0<br>00000 | ADD<br>100000 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** ADD rd, rs, rt **MIPS32 (MIPS I)**

**Purpose:**

To add 32-bit integers. If an 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, the destination register is not modified and an Integer Overflow exception occurs.

- If the addition does not overflow, the 32-bit result is signed-extended and placed into GPR *rd*.

**Restrictions:**

If either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

**Operation:**

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← (GPR[rs]_31||GPR[rs]_31..0) + (GPR[rt]_31||GPR[rt]_31..0)
if temp_32 ≠ temp_31 then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← sign_extend(temp_31..0)
endif
```

**Exceptions:**

Integer Overflow

**Programming Notes:**

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

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| COP1 010001 | | fmt | | ft | | fs | | fd | | ADD 000000 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:**   ADD.S fd, fs, ft                                       **MIPS32 (MIPS I)**
          ADD.D fd, fs, ft                                       **MIPS32 (MIPS I)**
          ADD.PS fd, fs, ft                                      **MIPS64 (MIPS V)**

**Purpose:**

To add floating point 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 by using to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. ADD.PS adds the upper and lower halves of FPR *fs* and FPR *ft* independently, and ORs together any generated exceptions.

*Cause* bits are ORed into the *Flag* bits if no exception is taken.

**Restrictions:**

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*. If they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of ADD.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

    StoreFPR (fd, fmt, ValueFPR(fs, fmt) +$_{fmt}$ ValueFPR(ft, fmt))

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow

| 31           26 | 25        21 | 20        16 | 15                  0 |
|---|---|---|---|
| ADDI<br>001000 | rs | rt | immediate |
| 6 | 5 | 5 | 16 |
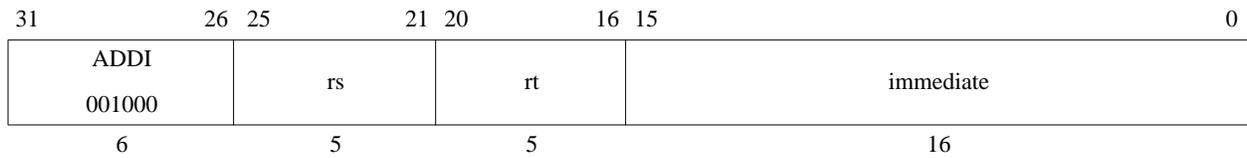
**Format:** ADDI rt, rs, immediate                                         **MIPS32 (MIPS I)**

**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, the destination register is not modified and an Integer Overflow exception occurs.

- If the addition does not overflow, the 32-bit result is sign-extended and placed into GPR *rt*.

**Restrictions:**

If GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

**Operation:**

```
if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif
temp ← (GPR[rs]₃₁||GPR[rs]₃₁..₀) + sign_extend(immediate)
if temp₃₂ ≠ temp₃₁ 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.

## Add Immediate Unsigned Word

**ADDIU**

| 31        26 | 25        21 | 20        16 | 15                          0 |
|--------------|--------------|--------------|-------------------------------|
| ADDIU<br>001001 | rs | rt | immediate |
| 6 | 5 | 5 | 16 |
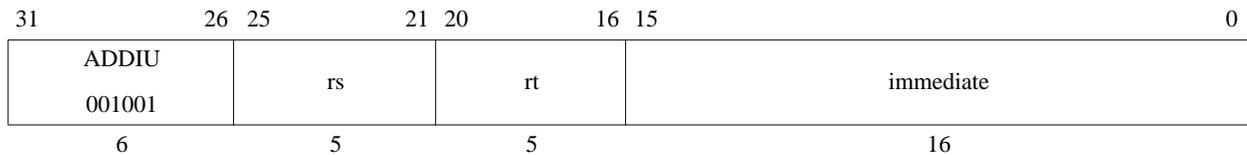
**Format:** ADDIU rt, rs, immediate

**MIPS32 (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 sign-extended and placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**

If GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

**Operation:**

```
if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt]← sign_extend(temp_{31..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. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

## Add Unsigned Word                                                          ADDU

| 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 |
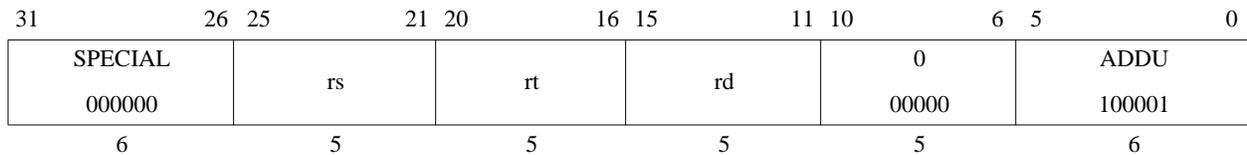
**Format:** ADDU rd, rs, rt                                           **MIPS32 (MIPS I)**

**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 sign-extended and placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**


If either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

**Operation:**

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← sign_extend(temp_{31..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. This instruction is appropriate for unsigned arithmetic, 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COP1X 010011 | | rs | | ft | | fs | | fd | | ALNV.PS 011110 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:** ALNV.PS fd, fs, ft, rs **MIPS64 (MIPS V)**

**Purpose:**

To align a misaligned pair of paired single values

**Description:** fd ← ByteAlign(rs$_{2..0}$, fs, ft)

FPR *fs* is concatenated with FPR *ft* and this value is funnel-shifted by GPR $rs_{2..0}$ bytes, and written into FPR *fd*. If GPR $rs_{2..0}$ is 0, *fd* receives *fs*. If GPR $rs_{2..0}$ is 4, the operation depends on the current endianness.

Figure 3-1 illustrates the following example: for a big-endian operation and a byte alignment of 4, the upper half of *fd* receives the lower half of the paired single value in *fs*, and the lower half of *fd* receives the upper half of the paired single value in *ft*.

**Figure 3-1 Example of an ALNV.PS Operation**
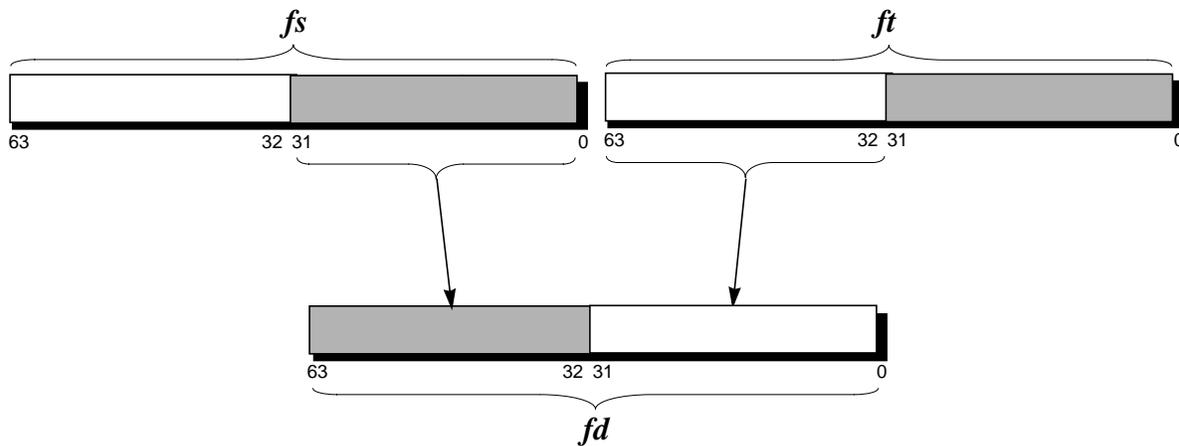


The move is nonarithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields *fs, ft,* and *fd* must specify FPRs valid for operands of type *PS*. If they are not valid, the result is **UNPRE-DICTABLE**.

If GPR $rs_{1..0}$ are non-zero, the results are **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
if GPR[rs]2..0 = 0 then
    StoreFPR(fd, PS,ValueFPR(fs,PS))
else if GPR[rs]2..0 ≠ 4 then
    UNPREDICTABLE
else if BigEndianCPU then
    StoreFPR(fd, PS, ValueFPR(fs, PS)31..0 || ValueFPR(ft,PS)63..32)
else
    StoreFPR(fd, PS, ValueFPR(ft, PS)31..0 || ValueFPR(fs,PS)63..32)
endif
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Programming Notes:**

ALNV.PS is designed to be used with LUXC1 to load 8 bytes of data from any 4-byte boundary. For example:

```
/* Copy T2 bytes (a multiple of 16) of data T0 to T1, T0 unaligned, T1 aligned.
           Reads one dw beyond the end of T0. */
    LUXC1    F0, 0(T0) /* set up by reading 1st src dw */
    LI       T3, 0    /* index into src and dst arrays */
    ADDIU    T4, T0, 8 /* base for odd dw loads */
    ADDIU    T5, T1, -8/* base for odd dw stores */
LOOP:
    LUXC1    F1, T3(T4)
    ALNV.PS  F2, F0, F1, T0/* switch F0, F1 for little-endian */
    SDC1     F2, T3(T1)
    ADDIU    T3, T3, 16
    LUXC1    F0, T3(T0)
    ALNV.PS  F2, F1, F0, T0/* switch F1, F0 for little-endian */
    BNE      T3, T2, LOOP
    SDC1     F2, T3(T5)
DONE:
```

ALNV.PS is also useful with SUXC1 to store paired-single results in a vector loop to a possibly misaligned address:

```
/* T1[i] =   T0[i] + F8, T0 aligned, T1 unaligned. */
            CVT.PS.S F8, F8, F8/* make addend paired-single */

/* Loop header computes 1st pair into F0, stores high half if T1 */
/* misaligned */

LOOP:
    LDC1      F2, T3(T4)/* get T0[i+2]/T0[i+3] */
    ADD.PS    F1, F2, F8/* compute T1[i+2]/T1[i+3] */
    ALNV.PS   F3, F0, F1, T1/* align to dst memory */
    SUXC1     F3, T3(T1)/* store to T1[i+0]/T1[i+1] */
    ADDIU     T3, 16    /* i = i + 4 */
    LDC1      F2, T3(T0)/* get T0[i+0]/T0[i+1] */
    ADD.PS    F0, F2, F8/* compute T1[i+0]/T1[i+1] */
    ALNV.PS   F3, F1, F0, T1/* align to dst memory */
    BNE       T3, T2, LOOP
    SUXC1     F3, T3(T5)/* store to T1[i+2]/T1[i+3] */

/* Loop trailer stores all or half of F0, depending on T1 alignment */
```

## And

**AND**

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| SPECIAL<br>000000 | | rs | | rt | | rd | | 0<br>00000 | | AND<br>100100 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:**   AND rd, rs, rt     **MIPS32 (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<br>001100 | rs | rt | immediate |
| 6 | 5 | 5 | 16 |
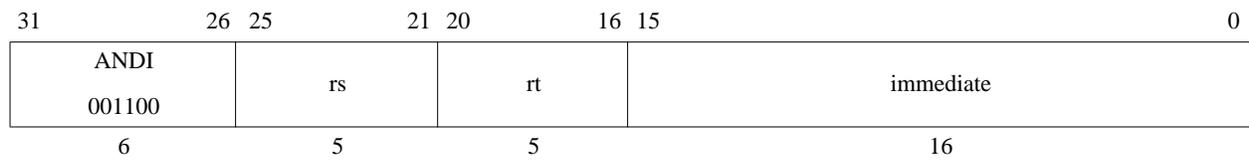
**Format:** ANDI rt, rs, immediate          **MIPS32 (MIPS I)**

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

**Exceptions:**

None

| | | | |
|---|---|---|---|
| **Unconditional Branch** | | | **B** |

```
31            26 25        21 20        16 15                            0
┌─────────────┬───────────┬───────────┬─────────────────────────────────┐
│    BEQ      │     0     │     0     │                                 │
│             │           │           │            offset               │
│   000100    │   00000   │   00000   │                                 │
└─────────────┴───────────┴───────────┴─────────────────────────────────┘
      6             5           5                     16
```
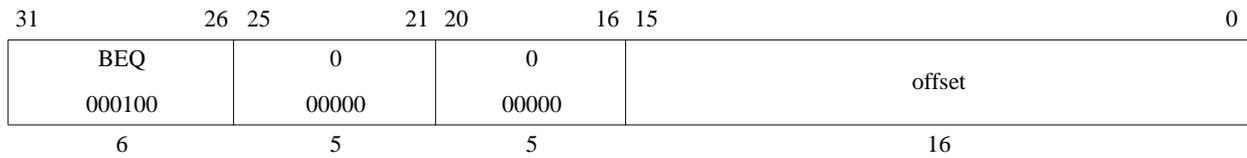
**Format:** `B offset`                                                                          **Assembly Idiom**

**Purpose:**

To do an unconditional branch

**Description:** `branch`

B offset is the assembly idiom used to denote an unconditional branch. The actual instruction is interpreted by the hardware as BEQ r0, r0, offset.

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.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:     target_offset ← sign_extend(offset || 0²)
I+1:   PC ← PC + target_offset
```

**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 addresses outside this range.

| 31 26 | 25 21 | 20 16 | 15 0 |
|---|---|---|---|
| REGIMM | 0 | BGEZAL | offset |
| 000001 | 00000 | 10001 | |
| 6 | 5 | 5 | 16 |

**Format:** `BAL rs, offset` **Assembly Idiom**

**Purpose:**

To do an unconditional PC-relative procedure call

**Description:** `procedure_call`

BAL offset is the assembly idiom used to denote an unconditional branch. The actual instruction is iterpreted by the hardware as BGEZAL r0, offset.

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues 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.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

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

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
        GPR[31] ← PC + 8
I+1:    PC ← PC + target_offset
```

**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 addresses outside this range.

| 31 | 26 | 25 | 21 | 20 | 18 | 17 | 16 | 15 | 0 |
|----|----|----|----|----|----|----|----|----|---|
| COP1 010001 | | BC 01000 | | cc | | nd 0 | tf 0 | offset | |
| 6 | | 5 | | 3 | | 1 | 1 | 16 | |

**Format:** BC1F    offset (cc = 0 implied)                                          **MIPS32 (MIPS I)**
          BC1F    cc, offset                                                        **MIPS32 (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), the program branches 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.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On 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*.

```
I:      condition ← FPConditionCode(cc) = 0
        target_offset ← (offset₁₅)^GPRLEN-(16+2) || offset || 0²
I+1:    if condition then
            PC ← PC + target_offset
        endif
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

*Floating Point Exceptions:*

Unimplemented Operation

**Programming Notes:**

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

**Historical Information:**

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/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 and MIPS32 architectures add 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 and MIPS32.

In the MIPS I, II, and III architectures there must be at least one instruction between the compare instruction that sets the condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.

| 31 | 26 | 25 | 21 | 20 | 18 | 17 | 16 | 15 | 0 |
|----|----|----|----|----|----|----|----|----|----|
| COP1 010001 | | BC 01000 | | cc | | nd 1 | tf 0 | offset | |
| 6 | | 5 | | 3 | | 1 | 1 | 16 | |

**Format:**  BC1FL   offset (cc = 0 implied)                                 **MIPS32 (MIPS II)**
           BC1FL   cc, offset                                                      **MIPS32 (MIPS IV)**

**Purpose:**

To test an FP condition code and make a PC-relative conditional branch; execute the instruction in 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), the program branches 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.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On 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*.

```
I:      condition ← FPConditionCode(cc) = 0
        target_offset ← (offset_15)^GPRLEN-(16+2) || offset || 0^2
I+1:    if condition then
            PC ← PC + target_offset
        else
            NullifyCurrentInstruction()
        endif
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation

**Programming Notes:**

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

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BC1F instruction instead.

**Historical Information:**

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/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 and MIPS32 architectures add 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 and MIPS32.

In the MIPS II andIII architectionrs 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.

| 31 | 26 | 25 | 21 | 20 | 18 | 17 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| COP1 010001 | | BC 01000 | | cc | | nd 0 | tf 1 | offset | |
| 6 | | 5 | | 3 | | 1 | 1 | 16 | |

**Format:**  BC1T offset (cc = 0 implied)                                    **MIPS32 (MIPS I)**
          BC1T cc, offset                                             **MIPS32 (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), the program branches 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.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On 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*.

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

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

*Floating Point Exceptions:*

Unimplemented Operation

**Programming Notes:**

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

**Historical Information:**

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/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 and MIPS32 architectures add 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 and MIPS32.

In the MIPS I, II, and III architectures **t**here must be at least one instruction between the compare instruction that sets the condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.

| 31 | 26 | 25 | 21 | 20 | 18 | 17 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| COP1 010001 | | BC 01000 | | cc | | nd 1 | tf 1 | offset | |
| 6 | | 5 | | 3 | | 1 | 1 | 16 | |

**Format:**   BC1TL    offset (cc = 0 implied)                        **MIPS32 (MIPS II)**
             BC1TL    cc, offset                                      **MIPS32 (MIPS IV)**

**Purpose:**

To test an FP condition code and do a PC-relative conditional branch; execute the instruction in 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), the program branches 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.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.
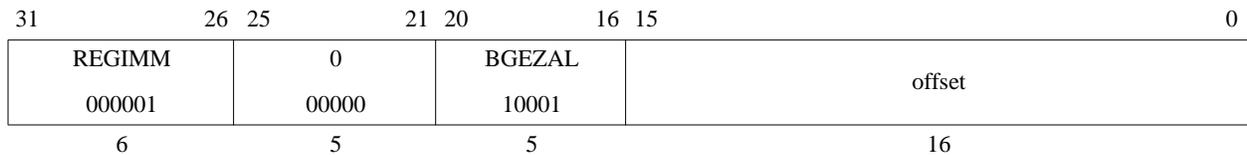
**Operation:**

This operation specification is for the general Branch On 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*.

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

$$\text{I:} \quad condition \leftarrow FPConditionCode(cc) = 1$$
$$target\_offset \leftarrow (offset_{15})^{GPRLEN-(16+2)} \;||\; offset \;||\; 0^2$$

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation

**Programming Notes:**

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

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BC1T instruction instead.

**Historical Information:**

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/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 and MIPS32 architectures add 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 and MIPS32.

In the MIPS II andIII architectionrs 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.

| 31 | 26 | 25 | 21 | 20 | 18 | 17 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| COP2<br>010010 | | BC<br>01000 | | cc | | nd<br>0 | tf<br>0 | offset | |
| 6 | | 5 | | 3 | | 1 | 1 | 16 | |

**Format:**   BC2F    offset (cc = 0 implied)             **MIPS32 (MIPS I)**
                BC2F    cc, offset                      **MIPS32 (MIPS IV)**

**Purpose:**

To test a COP2 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 COP2 condition specified by *CC* is false (0), the program branches to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for *tf* and *nd*.

```
I:      condition ← COP2Condition(cc) = 0
        target_offset ← (offset₁₅)^GPRLEN-(16+2) || offset || 0²
I+1:    if condition then
            PC ← PC + target_offset
        endif
```

$$\text{target\_offset} \leftarrow (\text{offset}_{15})^{\text{GPRLEN}-(16+2)} \; || \; \text{offset} \; || \; 0^2$$

**Exceptions:**

Coprocessor Unusable, 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 addresses outside this range.

| 31 | 26 | 25 | 21 | 20 | 18 | 17 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| COP2 010010 | | BC 01000 | | cc | | nd 1 | tf 0 | offset | |
| 6 | | 5 | | 3 | | 1 | 1 | 16 | |

**Format:** BC2FL    offset (cc = 0 implied)          **MIPS32 (MIPS II)**
        BC2FL    cc, offset                        **MIPS32 (MIPS IV)**

**Purpose:**

To test a COP2 condition code and make a PC-relative conditional branch; execute the instruction in 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 COP2 condition specified by *CC* is false (0), the program branches 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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for *tf* and *nd*.

```
I:      condition ← COP2Condition(cc) = 0
        target_offset ← (offset₁₅)^(GPRLEN-(16+2)) || offset || 0²
I+1:    if condition then
            PC ← PC + target_offset
        else
            NullifyCurrentInstruction()
        endif
```

**Exceptions:**

Coprocessor Unusable, 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 addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BC2F instruction instead.

| 31 | 26 | 25 | 21 | 20 | 18 | 17 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| COP2 010010 | | BC 01000 | | cc | | nd 0 | tf 1 | offset | |
| 6 | | 5 | | 3 | | 1 | 1 | 16 | |

**Format:**  BC2T offset (cc = 0 implied)                    **MIPS32 (MIPS I)**
             BC2T cc, offset                                  **MIPS32 (MIPS IV)**

**Purpose:**

To test a COP2 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 COP2 condition specified by *CC* is true (1), the program branches to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for *tf* and *nd*.

```
I:      condition ← COP2Condition(cc) = 1
        target_offset ← (offset₁₅)^GPRLEN-(16+2) || offset || 0²
I+1:    if condition then
            PC ← PC + target_offset
        endif
```

**Exceptions:**

Coprocessor Unusable, 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 addresses outside this range.

| 31 | 26 | 25 | 21 | 20 | 18 | 17 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| COP2 010010 | | BC 01000 | | cc | | nd 1 | tf 1 | offset | |
| 6 | | 5 | | 3 | | 1 | 1 | 16 | |

**Format:** BC2TL  offset (cc = 0 implied)                    **MIPS32 (MIPS II)**
          BC2TL  cc, offset                             **MIPS32 (MIPS IV)**

**Purpose:**

To test a COP2 condition code and do a PC-relative conditional branch; execute the instruction in 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 COP2 condition specified by *CC* is true (1), the program branches 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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for *tf* and *nd*.

```
I:      condition ← COP2Condition(cc) = 1
        target_offset ← (offset₁₅)^GPRLEN-(16+2) || offset || 0²
I+1:    if condition then
            PC ← PC + target_offset
        else
            NullifyCurrentInstruction()
        endif
```

**Exceptions:**

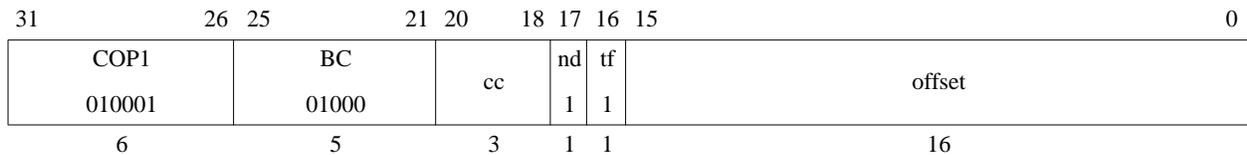Coprocessor Unusable, Reserved Instruction

**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 addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BC2T instruction instead.

**Branch on Equal**                                                    BEQ

| 31          26 | 25        21 | 20      16 | 15                                    0 |
|----------------|--------------|------------|-----------------------------------------|
| BEQ<br>000100  | rs           | rt         | offset                                  |
| 6              | 5            | 5          | 16                                      |
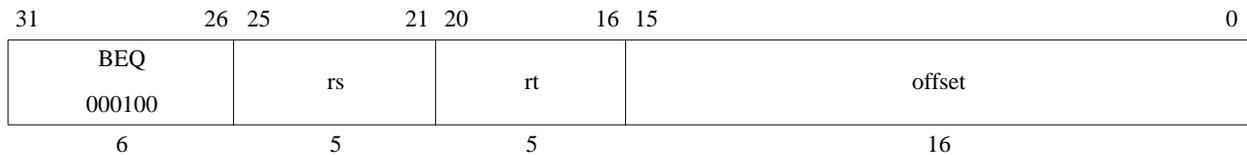
**Format:** `BEQ rs, rt, offset`                                    **MIPS32 (MIPS I)**

**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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
            condition ← (GPR[rs] = GPR[rt])
I+1:    if condition then
            PC ← PC + target_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 addresses outside this range.

BEQ r0, r0 offset, expressed as B offset, is the assembly idiom used to denote an unconditional branch.

62                        MIPS64™ Architecture For Programmers Volume II, Revision 0.95

**Branch on Equal Likely**                                                                                      **BEQL**

| 31        26 | 25        21 | 20        16 | 15                        0 |
|--------------|--------------|--------------|-----------------------------|
| BEQL<br>010100 | rs | rt | offset |
| 6 | 5 | 5 | 16 |

**Format:** `BEQL rs, rt, offset`                                                          **MIPS32 (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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
            condition ← (GPR[rs] = GPR[rt])
I+1:    if condition then
            PC ← PC + target_offset
        else
            NullifyCurrentInstruction()
        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 addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BEQ instruction instead.

**Historical Information:**

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.

| 31 26 | 25 21 | 20 16 | 15 0 |
|---|---|---|---|
| REGIMM<br>000001 | rs | BGEZ<br>00001 | offset |
| 6 | 5 | 5 | 16 |
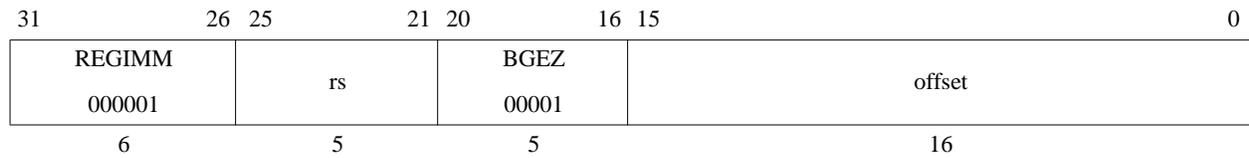
**Format:** BGEZ rs, offset **MIPS32 (MIPS I)**

**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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
        condition ← GPR[rs] ≥ 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_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 addresses outside this range.

**Branch on Greater Than or Equal to Zero and Link**                                      **BGEZAL**

| 31 26 | 25 21 | 20 16 | 15 0 |
|---|---|---|---|
| REGIMM<br>000001 | rs | BGEZAL<br>10001 | offset |
| 6 | 5 | 5 | 16 |

**Format:**  BGEZAL rs, offset                                                       **MIPS32 (MIPS I)**

**Purpose:**

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

**Description:** if rs $\geq$ 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 continues 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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

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

**Operation:**

```
I:     target_offset ← sign_extend(offset || 0²)
       condition ← GPR[rs] ≥ 0^GPRLEN
       GPR[31] ← PC + 8
I+1:   if condition then
           PC ← PC + target_offset
       endif
```

**Exceptions:**

None

**Programming Notes:**

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

BGEZAL r0, offset, expressed as BAL offset, is the assembly idiom used to denote a PC-relative branch and link. BAL is used in a manner similar to JAL, but provides PC-relative addressing and a more limited target PC range.

| 31          26 | 25          21 | 20          16 | 15                              0 |
|----------------|----------------|----------------|-----------------------------------|
| REGIMM<br>000001 | rs | BGEZALL<br>10011 | offset |
| 6 | 5 | 5 | 16 |

**Format:** `BGEZALL rs, offset`                                          **MIPS32 (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 continues 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 reexecuted. The result of executing such an instruction is **UNPREDICTABLE**. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
        condition ← GPR[rs] ≥ 0^GPRLEN
        GPR[31] ← PC + 8
I+1:    if condition then
            PC ← PC + target_offset
        else
            NullifyCurrentInstruction()
        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 addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BGEZAL instruction instead.

**Historical Information:**

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.

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

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

**Format:** `BGEZL rs, offset`                                                       **MIPS32 (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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
        condition ← GPR[rs] ≥ 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_offset
        else
            NullifyCurrentInstruction()
        endif
```

**Exceptions:**

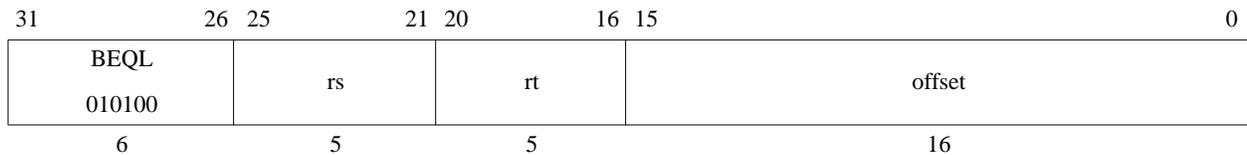**None**

**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 addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BGEZ instruction instead.

**Historical Information:**

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.

| Branch on Greater Than Zero | | | | BGTZ |
|---|---|---|---|---|

```
31          26 25        21 20        16 15                        0
+-------------+-----------+-------------+------------------------------+
|    BGTZ     |           |      0      |                              |
|             |    rs     |             |            offset            |
|   000111    |           |    00000    |                              |
+-------------+-----------+-------------+------------------------------+
      6             5            5                     16
```

**Format:** BGTZ rs, offset                                                **MIPS32 (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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
        condition ← GPR[rs] > 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_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 addresses outside this range.

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

**Format:** BGTZL rs, offset **MIPS32 (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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
        condition ← GPR[rs] > 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_offset
        else
            NullifyCurrentInstruction()
        endif
```
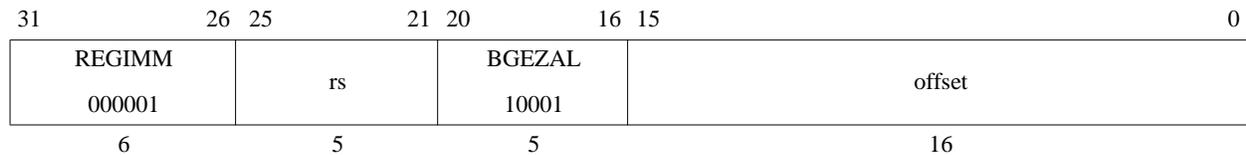
**Exceptions:**

**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 addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BGTZ instruction instead.

**Historical Information:**

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.

## Branch on Less Than or Equal to Zero BLEZ

| 31      | 26 25 | 21 20        | 16 15      | 0 |
|---------|-------|--------------|------------|---|
| BLEZ<br>000110 | rs | 0<br>00000 | offset |   |
| 6       | 5     | 5            | 16         |   |

**Format:** BLEZ rs, offset                                                       **MIPS32 (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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
        condition ← GPR[rs] ≤ 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_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 addresses outside this range.

| 31 | | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|----|----|----|----|----|----|----|----|----|
| BLEZL<br>010110 | | | rs | | 0<br>00000 | | offset | |
| 6 | | | 5 | | 5 | | 16 | |

**Format:** BLEZL rs, offset                                                    **MIPS32 (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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
        condition ← GPR[rs] ≤ 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_offset
        else
            NullifyCurrentInstruction()
        endif
```
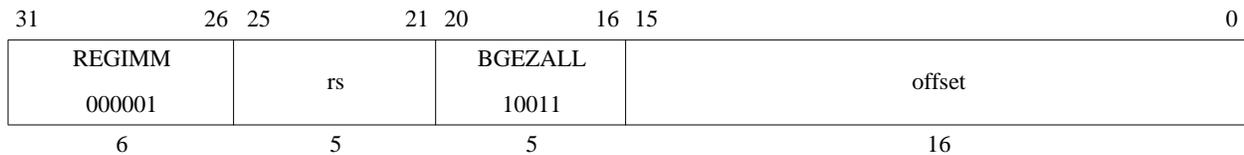
**Exceptions:**

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 addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BLEZ instruction instead.

**Historical Information:**

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.

**Branch on Less Than Zero**                                                                                          **BLTZ**

| 31          26 | 25          21 | 20          16 | 15                                              0 |
|----------------|----------------|----------------|---------------------------------------------------|
| REGIMM<br>000001 | rs | BLTZ<br>00000 | offset |
| 6 | 5 | 5 | 16 |

**Format:** `BLTZ rs, offset`                                                                        **MIPS32 (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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**
```
I:      target_offset ← sign_extend(offset || 0²)
        condition ← GPR[rs] < 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_offset
        endif
```

**Exceptions:**
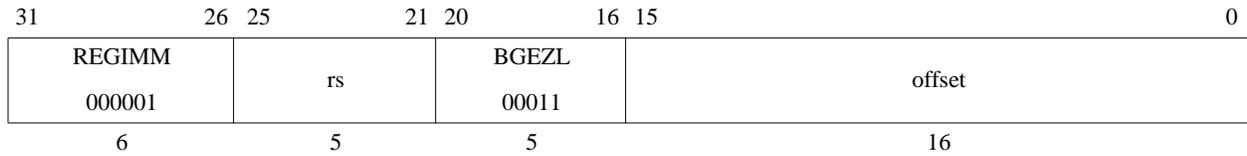
None

**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 addresses outside this range.

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

**Format:** BLTZAL rs, offset **MIPS32 (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 continues 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 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 reexecuted. The result of executing such an instruction is UNPREDICTABLE. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:     target_offset ← sign_extend(offset || 0²)
       condition ← GPR[rs] < 0^GPRLEN
       GPR[31] ← PC + 8
I+1:   if condition then
           PC ← PC + target_offset
       endif
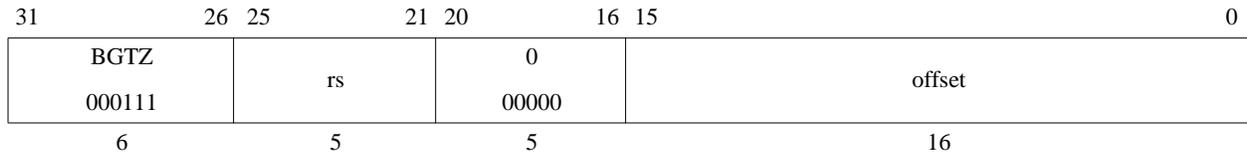```

**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 addresses outside this range.

## Branch on Less Than Zero and Link Likely — BLTZALL

| 31          | 26 25 | 21 20           | 16 15  | 0 |
|-------------|-------|-----------------|--------|---|
| REGIMM<br>000001 | rs | BLTZALL<br>10010 | offset |   |
| 6 | 5 | 5 | 16 | |

**Format:** BLTZALL rs, offset                                      **MIPS32 (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 continues 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 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 reexecuted. The result of executing such an instruction is UNPREDICTABLE. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:     target_offset ← sign_extend(offset || 0²)
       condition ← GPR[rs] < 0^GPRLEN
       GPR[31] ← PC + 8
I+1:   if condition then
           PC ← PC + target_offset
       else
           NullifyCurrentInstruction()
       endif
```
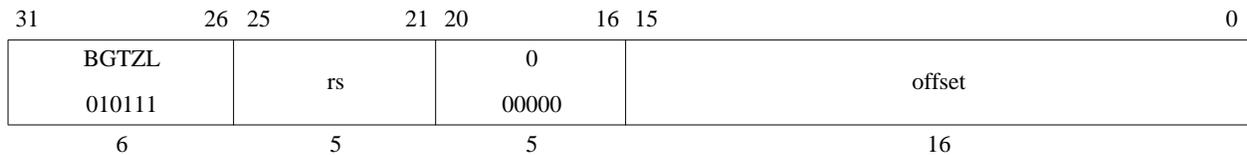
**Exceptions:**

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 addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BLTZAL instruction instead.

**Historical Information:**

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.

**Branch on Less Than Zero Likely**                                                    **BLTZL**

| 31          26 | 25        21 | 20        16 | 15                            0 |
|----------------|--------------|--------------|---------------------------------|
| REGIMM<br>000001 | rs | BLTZL<br>00010 | offset |
| 6 | 5 | 5 | 16 |
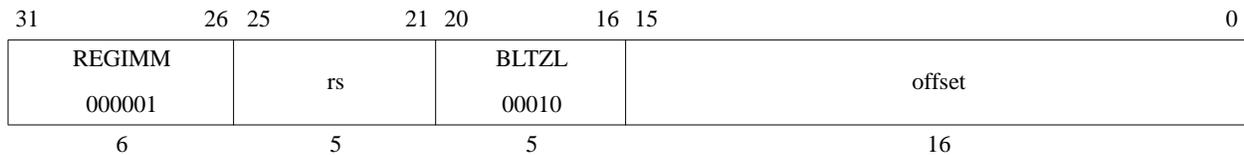
**Format:** `BLTZL rs, offset`                                          **MIPS32 (MIPS II)**

**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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
        condition ← GPR[rs] < 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_offset
        else
            NullifyCurrentInstruction()
        endif
```

**Exceptions:**
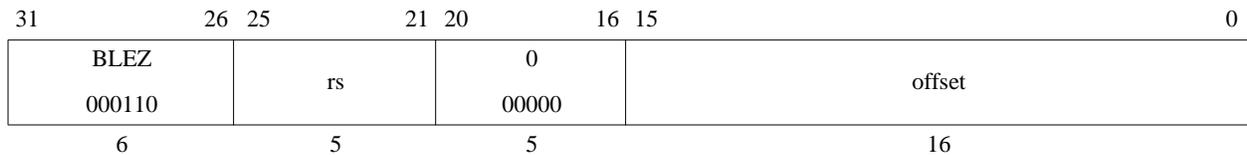
None

**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 addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BLTZ instruction instead.

**Historical Information:**

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.

**Branch on Not Equal**                                                              **BNE**

| 31          26 | 25          21 | 20          16 | 15                                    0 |
|----------------|----------------|----------------|-----------------------------------------|
| BNE<br>000101  | rs             | rt             | offset                                  |
| 6              | 5              | 5              | 16                                      |
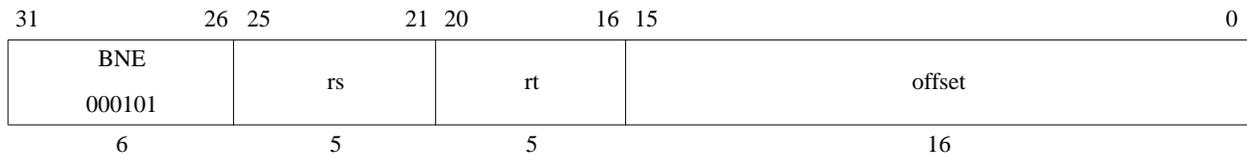
**Format:** BNE rs, rt, offset                                          **MIPS32 (MIPS I)**

**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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
        condition ← (GPR[rs] ≠ GPR[rt])
I+1:    if condition then
            PC ← PC + target_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 addresses outside this range.

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|
| BNEL 010101 | | rs | | rt | | offset | |
| 6 | | 5 | | 5 | | 16 | |

**Format:** BNEL rs, rt, offset <span style="float:right">**MIPS32 (MIPS II)**</span>

**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:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:      target_offset ← sign_extend(offset || 0²)
        condition ← (GPR[rs] ≠ GPR[rt])
I+1:    if condition then
            PC ← PC + target_offset
        else
            NullifyCurrentInstruction()
        endif
```

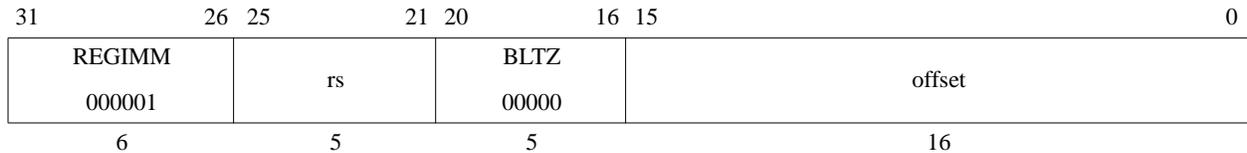**Exceptions:**

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 addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BNE instruction instead.

**Historical Information:**

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.

| 31 | 26 | 25 | 6 | 5 | 0 |
|---|---|---|---|---|---|
| SPECIAL<br>000000 | | code | | BREAK<br>001101 | |
| 6 | | 20 | | 6 | |
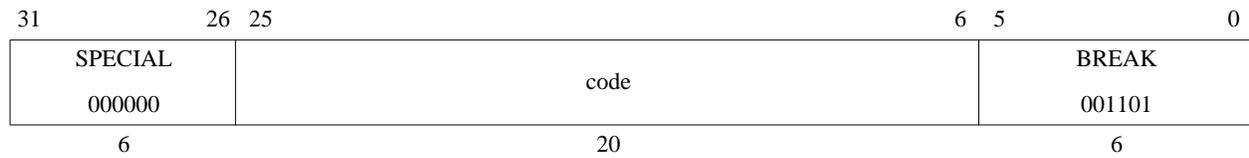
**Format:** BREAK **MIPS32 (MIPS I)**

**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

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| COP1 010001 | | fmt | | ft | | fs | | cc | | 0 | A 0 | FC 11 | | cond | |
| 6 | | 5 | | 5 | | 5 | | 3 | | 1 | 1 | 2 | | 4 | |

**Format:**  C.cond.S fs, ft (cc = 0 implied)                                **MIPS32 (MIPS I)**
C.cond.D fs, ft (cc = 0 implied**)**                                   **MIPS32 (MIPS I)**
C.cond.PS fs, ft(cc = 0 implied)                                      **MIPS64 (MIPS V)**
C.cond.S cc, fs, ft                                                   **MIPS32 (MIPS IV)**
C.cond.D cc, fs, ft                                                   **MIPS32 (MIPS IV)**
C.cond.PS cc, fs, ft                                                  **MIPS64 (MIPS V)**

**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, the result is true; otherwise, the result is false. If no exception is taken, the result is written into condition code *CC*; true is 1 and false is 0.

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

If one of the values is an SNaN, or $cond_3$ is set and at least one of the values is a QNaN, an Invalid Operation condition is raised and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written and an Invalid Operation exception is taken immediately. Otherwise, the Boolean result is written into condition code *CC*.

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 among the 16 comparison predicates. The Boolean result of the instruction is obtained by substituting the Boolean value of each ordering relation for the two FP values in the equation. If the *equal* relation is true, for example, then all four example predicates above yield a true result. If the *unordered* relation is true then only the final predicate, *unordered or equal*, yields a true result.

Logical negation of a compare result allows eight distinct comparisons to test for the 16 predicates as shown in . 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, and the second predicate must be false, and vice versa. (Note that the False predicate is never true and False/True do not follow the normal pattern.)

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 can be made with the Branch on FP True (BC1T) instruction and the truth of the second can be made with Branch on FP False (BC1F).

Table 3-24 shows another set of eight compare operations, distinguished by a $cond_3$ value of 1 and testing the same 16 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*, an Invalid Operation exception occurs.

**Table 3-24 FPU Comparisons Without Special Operand Exceptions**

| Instruction | Comparison Predicate | | | | | | Comparison CC Result | | Instruction | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Cond Mnemonic** | **Name of Predicate and Logically Negated Predicate (Abbreviation)** | **Relation Values** | | | | **If Predicate Is True** | **Inv Op Excp. if QNaN ?** | | **Condition Field** | |
| | | **>** | **<** | **=** | **?** | | | | **3** | **2..0** |
| F | False [this predicate is always False] | F | F | F | F | F | No | | 0 | 0 |
| | True (T) | 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 | | | | | | | | | | |

**Table 3-25 FPU Comparisons With Special Operand Exceptions for QNaNs**

| Instruction | Comparison Predicate | | | | | | Comparison CC Result | | Instruction | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Cond Mnemonic** | **Name of Predicate and Logically Negated Predicate (Abbreviation)** | **Relation Values** | | | | **If Predicate Is True** | **Inv Op Excp If QNaN?** | | **Condition 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* | | | | | | | | | | |

**Restrictions:**

The fields *fs* and *ft* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICT-ABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of C.cond.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode, or if the condition code number is odd.

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

For c.cond.PS, the pseudo code above is repeated for both halves of the operand registers, treating each half as an independent single-precision values. Exceptions on the two halves are logically ORed and reported together. The results of the lower half comparison are written to condition code CC; the results of the upper half comparison are written to condition code CC+1.

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation

**Programming Notes:**

FP computational instructions, including compare, that receive an operand value of Signaling NaN raise the Invalid Operation condition. 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
```

**Historical Information:**

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/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 and MIPS32 architectures add 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 malid for MIPS IV and MIPS32.

In the MIPS I, II, and III architectures there must be at least one instruction between the compare instruction that sets the condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|----|----|----|----|----|----|----|----|
| CACHE<br>101111 | | base | | op | | offset | |
| 6 | | 5 | | 5 | | 16 | |

**Format:** CACHE op, offset(base)                                                          **MIPS32**

**Purpose:**

To perform the cache operation specified by op.

**Description:**

The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

**Table 3-26 Usage of Effective Address**

| Operation Requires an | Type of Cache | Usage of Effective Address |
|-----------------------|---------------|----------------------------|
| Address | Virtual | The effective address is used to address the cache. It is implementation dependent whether an address translation is performed on the effective address (with the possibility that a TLB Refill or TLB Invalid exception might occur) |
| Address | Physical | The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache |
| Index | N/A | The effective address is translated by the MMU to a physical address. It is implementation dependent whether the effective address or the translated physical address is used to index the cache.<br><br>Assuming that the total cache size in bytes is CS, the associativity is A, and the number of bytes per tag is BPT, the following calculations give the fields of the address which specify the way and the index:<br><br>       OffsetBit $\leftarrow$ Log2(BPT)<br>       IndexBit $\leftarrow$ Log2(CS / A)<br>       WayBit $\leftarrow$ IndexBit + Ceiling(Log2(A))<br>       Way $\leftarrow$ Addr$_{\text{WayBit-1..IndexBit}}$<br>       Index $\leftarrow$ Addr$_{\text{IndexBit-1..OffsetBit}}$<br><br>For a direct-mapped cache, the Way calculation is ignored and the Index value fully specifies the cache tag. This is shown symbolically in the figure below. |

**Figure 3-2 Usage of Address Fields to Select Index and Way**
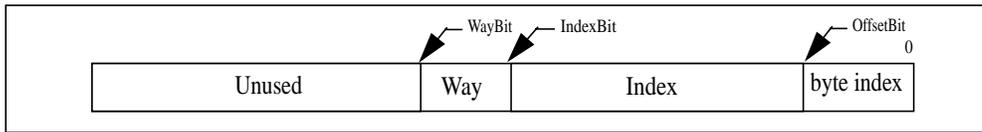


A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur on any operation. For index operations (where the address is used to index the cache but need not match the cache tag) software should use unmapped addresses to avoid TLB exceptions. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS, nor data Watch exceptions.

A Cache Error exception may occur as a byproduct of some operations performed by this instruction. For example, if a Writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error.

An Address Error Exception (with cause code equal AdEL) may occur if the effective address references a portion of the kernel address space which would normally result in such an exception. It is implementation dependent whether such an exception does occur.

It is implementation dependent whether a data watch is triggered by a cache instruction whose address matches the Watch register address match conditions.

Bits [17:16] of the instruction specify the cache on which to perform the operation, as follows:

**Table 3-27 Encoding of Bits[17:16] of CACHE Instruction**

| Code | Name | Cache |
|------|------|-------|
| 2#00 | I | Primary Instruction |
| 2#01 | D | Primary Data or Unified Primary |
| 2#10 | T | Tertiary |
| 2#11 | S | Secondary |

Bits [20:18] of the instruction specify the operation to perform. To provide software with a consistent base of cache operations, certain encodings must be supported on all processors. The remaining encodings are recommended.

**Table 3-28 Encoding of Bits [20:18] of the CACHE Instruction**

| Code | Caches | Name | Effective Address Operand Type | Operation | Compliance |
|---|---|---|---|---|---|
| 2#000 | I | Index Invalidate | Index | Set the state of the cache block at the specified index to invalid.<br><br>This required encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices. | Required |
| | D | Index Writeback Invalidate / Index Invalidate | Index | For a write-back cache: If the state of the cache block at the specified index is valid and dirty, write the block back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. | Required |
| | S, T | Index Writeback Invalidate / Index Invalidate | Index | For a write-through cache: Set the state of the cache block at the specified index to invalid.<br><br>This required encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. Note that Index Store Tag should be used to initialize the cache at powerup. | Optional |
| 2#001 | All | Index Load Tag | Index | Read the tag for the cache block at the specified index into the *TagLo* and *TagHi* Coprocessor 0 registers. If the *DataLo* and *DataHi* registers are implemented, also read the data corresponding to the byte index into the *DataLo* and *DataHi* registers.<br><br>The granularity and alignment of the data read into the *DataLo* and *DataHi* registers is implementation-dependent, but is typically the result of an aligned access to the cache, ignoring the appropriate low-order bits of the byte index. | Recommended |

**Table 3-28 Encoding of Bits [20:18] of the CACHE Instruction**

| Code | Caches | Name | Effective Address Operand Type | Operation | Compliance |
|---|---|---|---|---|---|
| 2#010 | All | Index Store Tag | Index | Write the tag for the cache block at the specified index from the *TagLo* and *TagHi* Coprocessor 0 registers.<br><br>This required encoding may be used by software to initialize the entire instruction of data caches by stepping through all valid indices. Doing so requires that the *TagLo* and *TagHi* registers associated with the cache be initialized first. | Required |
| 2#011 | All | Implementation Dependent | Unspecified | Available for implementation-dependent operation. | Optional |
| 2#100 | I, D | Hit Invalidate | Address | If the cache block contains the specified address, set the state of the cache block to invalid.<br><br>This required encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache. | Required (Instruction Cache Encoding Only), Recommended otherwise |
|  | S, T | Hit Invalidate | Address |  | Optional |
| 2#101 | I | Fill | Address | Fill the cache from the specified address. | Recommended |
|  | D | Hit Writeback Invalidate / Hit Invalidate | Address | For a write-back cache: If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid.<br><br>For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid.<br><br>This required encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache. | Required |
|  | S, T | Hit Writeback Invalidate / Hit Invalidate | Address |  | Optional |

**Table 3-28 Encoding of Bits [20:18] of the CACHE Instruction**

| Code | Caches | Name | Effective Address Operand Type | Operation | Compliance |
|------|--------|------|-------------------------------|-----------|------------|
| 2#110 | D | Hit Writeback | Address | If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state. For a write-through cache, this operation may be treated as a nop. | Recommended |
| | S, T | Hit Writeback | Address | | Optional |
| 2#111 | I, D | Fetch and Lock | Address | If the cache does not contain the specified address, fill it from memory, performing a writeback if required, and set the state to valid and locked. If the cache already contains the specified address, set the state to locked. In set-associative or fully-associative caches, the way selected on a fill from memory is implementation dependent.

The lock state may be cleared by executing an Index Invalidate, Index Writeback Invalidate, Hit Invalidate, or Hit Writeback Invalidate operation to the locked line, or via an Index Store Tag operation to the line that clears the lock bit. Note that clearing the lock state via Index Store Tag is dependent on the implementation-dependent cache tag and cache line organization, and that Index and Index Writeback Invalidate operations are dependent on cache line organization. Only Hit and Hit Writeback Invalidate operations are generally portable across implementations.

It is implementation dependent whether a locked line is displaced as the result of an external invalidate or intervention that hits on the locked line. Software must not depend on the locked line remaining in the cache if an external invalidate or intervention would invalidate the line if it were not locked.

It is implementation dependent whether a Fetch and Lock operation affects more than one line. For example, more than one line around the referenced address may be fetched and locked. It is recommended that only the single line containing the referenced address be affected. | Recommended |

MIPS64™ Architecture For Programmers Volume II, Revision 0.95

**Restrictions:**

The operation of this instruction is **UNDEFINED** for any operation/cache combination that is not implemented.

The operation of this instruction is **UNDEFINED** if the operaation requires an address, and that address is uncacheable.

**Operation:**

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, uncached) ← AddressTranslation(vAddr, DataReadReference)
CacheOp(op, vAddr, pAddr)
```

**Exceptions:**

TLB Refill Exception.

TLB Invalid Exception

Coprocessor Unusable Exception

Address Error Exception

Cache Error Exception

Bus Error Exception

**Fixed Point Ceiling Convert to Long Fixed Point**                    **CEIL.L.fmt**

| 31      26 | 25      21 | 20      16 | 15      11 | 10      6 | 5      0 |
|------------|------------|------------|------------|-----------|----------|
| COP1<br>010001 | fmt | 0<br>00000 | fs | fd | CEIL.L<br>001010 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**  CEIL.L.S    fd, fs                                    **MIPS64 (MIPS III)**
         CEIL.L.D    fd, fs                                    **MIPS64 (MIPS III)**

**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 and rounding toward $+\infty$ (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, an IEEE Invalid Operation condition exists, a d 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*.

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

    StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact, Overflow

| 31          26 | 25          21 | 20          16 | 15          11 | 10          6 | 5          0 |
|----------------|----------------|----------------|----------------|---------------|--------------|
| COP1<br>010001 | fmt | 0<br>00000 | fs | fd | CEIL.W<br>001110 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**  CEIL.W.S    fd, fs                                            **MIPS32 (MIPS II)**
          CEIL.W.D    fd, fs                                            **MIPS32 (MIPS II)**

**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 and rounding toward $+\infty$ (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, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *fd.*

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

**Operation:**

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact, Overflow

**Move Control Word From Floating Point**                                                    **CFC1**

| 31          26 | 25          21 | 20       16 | 15       11 | 10                          0 |
|----------------|----------------|-------------|-------------|-------------------------------|
| COP1<br>010001 | CF<br>00010    | rt          | fs          | 0<br>000 0000 0000            |
| 6              | 5              | 5           | 5           | 11                            |

**Format:**  CFC1 rt, fs                                                           **MIPS32 (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 to 64 bits.

**Restrictions:**

There are a few control registers defined for the floating point unit. The result is **UNPREDICTABLE** if *fs* specifies a register that does not exist.

**Operation:**

```
if fs = 0 then
    temp ← FIR
elseif fs = 25 then
    temp ← 0²⁴ || FCSR₃₁..₂₅ || FCSR₂₃
elseif fs = 26 then
    temp ← 0¹⁴ || FCSR₁₇..₁₂ || 0⁵ || FCSR₆..₂ || 0²
elseif fs = 28 then
    temp ← 0²⁰ || FCSR₁₁.₇ || 0⁴ || FCSR₂₄ || FCSR₁..₀
elseif fs = 31 then
    temp ← FCSR
else
    temp ← UNPREDICTABLE
endif
GPR[rt] ← sign_extend(temp)
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Historical Information:**

For the MIPS I, II and III architectures, the contents of GPR *rt* are **UNPREDICTABLE** for the instruction immediately following CFC1.

MIPS V and MIPS32 introduced the three control registers that access portions of FCSR. These registers were not available in MIPS I, II, III, or IV.

| 31          26 | 25          21 | 20          16 | 15          11 | 10                          0 |
|----------------|----------------|----------------|----------------|-------------------------------|
| COP2<br>010010 | CF<br>00010    | rt             | rd             | 0<br>000 0000 0000            |
| 6              | 5              | 5              | 5              | 11                            |
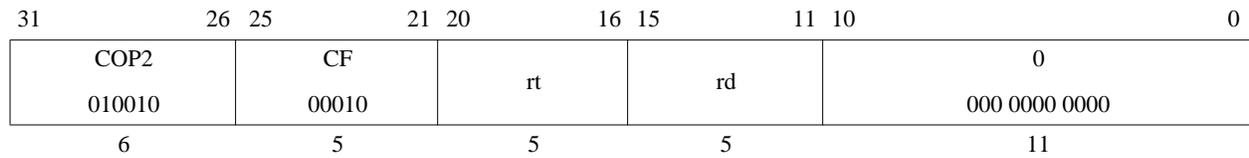
**Format:**  CFC2 rt, rd                                                                                 **MIPS32**

**Purpose:**

To copy a word from a Coprocessor 2 control register to a GPR

**Description:** rt ← CCR[2,rd]

Copy the 32-bit word from Coprocessor 2 control register *rd* into GPR *rt*, sign-extending it to 64 bits.

**Restrictions:**

The result is **UNPREDICTABLE** if *fs* specifies a register that does not exist.

**Operation:**

```
temp ← CCR[2,rd]
GPR[rt] ← sign_extend(temp)
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

## Count Leading Ones in Word                                                    CLO

| 31        26 | 25        21 | 20        16 | 15        11 | 10        6 | 5         0 |
|--------------|--------------|--------------|--------------|-------------|-------------|
| SPECIAL2     | rs           | rt           | rd           | 0           | CLO         |
| 011100       |              |              |              | 00000       | 100001      |
| 6            | 5            | 5            | 5            | 5           | 6           |

**Format:** CLO rd, rs                                                          MIPS32

**Purpose:**

**To** Count the number of leading ones in a word

**Description:** rd ← count_leading_ones rs

Bits 31..0 of GPR *rs* are scanned from most significant to least significant bit. The number of leading ones is counted and the result is written to GPR *rd*. If all of bits 31..0 were set in GPR *rs*, the result written to GPR *rd* is 32.

**Restrictions:**

To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the *rt* and *rd* fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values.

If GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

**Operation:**

```
if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif
temp ← 32
for i in 31 .. 0
    if GPR[rs]_i = 0 then
        temp ← 31 - i
        break
    endif
endfor
GPR[rd] ← temp
```

**Exceptions:**

None

| Count Leading Zeros in Word | | | | | CLZ |

**Count Leading Zeros in Word**                                                                          **CLZ**

| 31        26 | 25      21 | 20      16 | 15      11 | 10       6 | 5        0 |
|--------------|------------|------------|------------|------------|------------|
| SPECIAL2 | rs | rt | rd | 0 | CLZ |
| 011100 | | | | 00000 | 100000 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** CLZ rd, rs                                                                                   MIPS32

**Purpose**

Count the number of leading zeros in a word

**Description:** rd ← count_leading_zeros rs

Bits 31..0 of GPR *rs* are scanned from most significant to least significant bit. The number of leading zeros is counted and the result is written to GPR *rd*. If no bits were set in GPR *rs*, the result written to GPR *rt* is 32.

**Restrictions:**

To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in  both the *rt* and *rd* fields of the instruction.  The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values.

If GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

**Operation:**

```
if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif
temp ← 32
for i in 31 .. 0
    if GPR[rs]_i = 1 then
        temp ← 31 - i
        break
    endif
endfor
GPR[rd] ← temp
```

**Exceptions:**

None

| 31 | 26 | 25 | 24 | 0 |
|---|---|---|---|---|
| COP2 | | CO | cofun | |
| 010010 | | 1 | | |
| 6 | | 1 | 25 | |
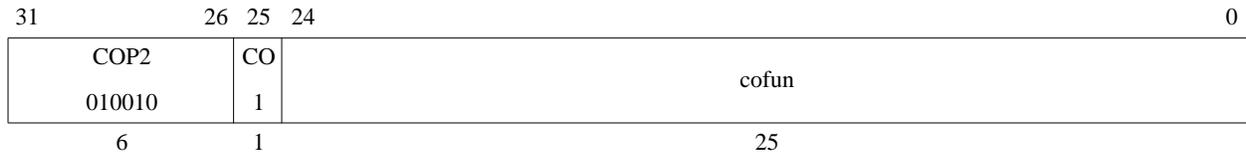
**Format:** `COP2 func` **MIPS32**

**Purpose:**

To performance an operation to Coprocessor 2

**Description:** `CoprocessorOperation(2, cofun)`

An implementation-dependent operation is performance to Coprocessor 2, with the *cofun* value passed as an argument. The operation may specify and reference internal coprocessor registers, and may change the state of the coprocessor conditions, but does not modify state within the processor. Details of coprocessor operation and internal state are described in the documentation for each Coprocessor 2 implementation.

**Restrictions:**

**Operation:**

```
CoprocessorOperation(2, cofun)
```

**Exceptions:**

Coprocessor Unusable
Reserved Instruction

## Move Control Word to Floating Point

**CTC1**

| 31 26 | 25 21 | 20 16 | 15 11 | 10 0 |
|---|---|---|---|---|
| COP1<br>010001 | CT<br>00110 | rt | fs | 0<br>000 0000 0000 |
| 6 | 5 | 5 | 5 | 11 |
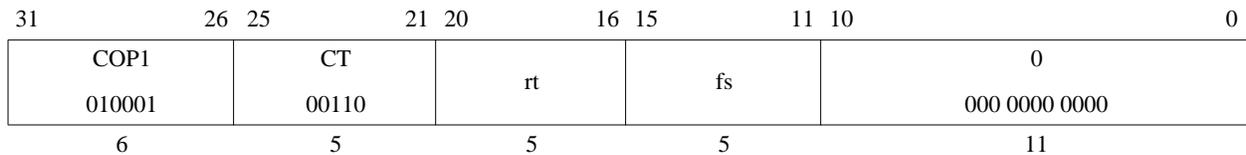
**Format:** CTC1   rt, fs                                                    **MIPS32 (MIPS I)**

**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 the FP (coprocessor 1) control register indicated by *fs*.

Writing to the floating point *Control/Status* register, the *FCSR*, causes the appropriate exception if any *Cause* bit and its corresponding *Enable* bit are both set. The register is written before the exception occurs. Writing to *FEXR* to set a cause bit whose enable bit is already set, or writing to *FENR* to set an enable bit whose cause bit is already set causes the appropriate exception. The register is written before the exception occurs.

**Restrictions:**

There are a few control registers defined for the floating point unit. The result is **UNPREDICTABLE** if *fs* specifies a register that does not exist.

**Operation:**

```
temp ← GPR[rt]₃₁..₀
if fs = 25 then
    if temp₃₁..₈ ≠ 0²⁴ then
        UNPREDICTABLE
    else
        FCSR ← temp₇..₁ || FCSR₂₄ || temp₀ || FCSR₂₂..₀
    endif
elseif fs = 26 then
    if temp₂₂..₁₈ ≠ 0 then
        UNPREDICTABLE
    else
        FCSR ← FCSR₃₁..₁₈ || temp₁₇..₁₂ || FCSR₁₁..₇ ||
        temp₆..₂ || FCSR₁..₀
    endif
elseif fs = 28 then
    if temp₂₂..₁₈ ≠ 0 then
        UNPREDICTABLE
    else
        FCSR ← FCSR₃₁..₂₅ || temp₂ || FCSR₂₃..₁₂ || temp₁₁..₇
        || FCSR₆..₂ || temp₁..₀
    endif
elseif fs = 31 then
    if temp₂₂..₁₈ ≠ 0 then
        UNPREDICTABLE
    else
        FCSR ← temp
    endif
else
    UNPREDICTABLE
endif
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation, Division-by-zero, Inexact, Overflow, Underflow

**Historical Information:**

For the MIPS I, II and III architectures, the contents of floating point control register *fs* are undefined for the instruction immediately following CTC1.

MIPS V and MIPS32 introduced the three control registers that access portions of FCSR. These registers were not available in MIPS I, II, III, or IV.

**Move Control Word to Coprocessor 2**                                                                        **CTC2**

| 31          26 | 25         21 | 20      16 | 15      11 | 10                                      0 |
|----------------|---------------|------------|------------|------------------------------------------|
| COP2           | CT            |            |            | 0                                        |
| 010010         | 00110         | rt         | rd         | 000 0000 0000                            |
| 6              | 5             | 5          | 5          | 11                                       |
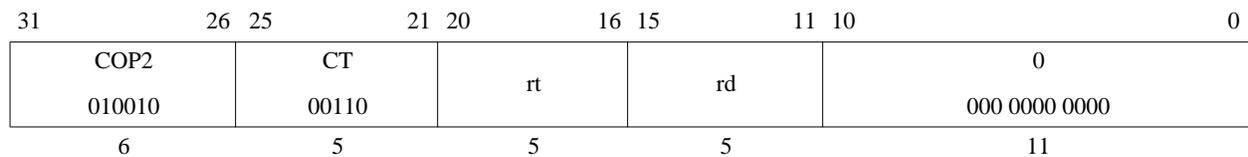
**Format:**   CTC2 rt, rd                                                                                     **MIPS32**

**Purpose:**

To copy a word from a GPR to a Coprocessor 2 control register

**Description:** CCR[2,rd] ← rt

Copy the low word from GPR *rt* into the Coprocessor 2control register indicated by *rd*.

**Restrictions:**

The result is **UNPREDICTABLE** if *rd* specifies a register that does not exist.

**Operation:**

```
temp ← GPR[rt]₃₁..₀
CCR[2,rd] ← temp
```

**Exceptions:**
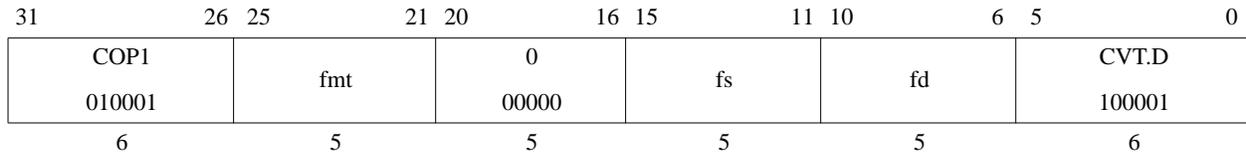
Coprocessor Unusable, Reserved Instruction

## Floating Point Convert to Double Floating Point

**CVT.D.fmt**

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| COP1<br>010001 | fmt | 0<br>00000 | fs | fd | CVT.D<br>100001 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**  CVT.D.S fd, fs            **MIPS32 (MIPS I)**
CVT.D.W fd, fs            **MIPS32 (MIPS I)**
CVT.D.L fd, fs            **MIPS64 (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 and 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—if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

**Operation:**

```
StoreFPR (fd, D, ConvertFmt(ValueFPR(fs, fmt), fmt, D))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact

**Floating Point Convert to Long Fixed Point** **CVT.L.fmt**

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COP1 010001 | | fmt | | 0 00000 | | fs | | fd | | CVT.L 100101 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |
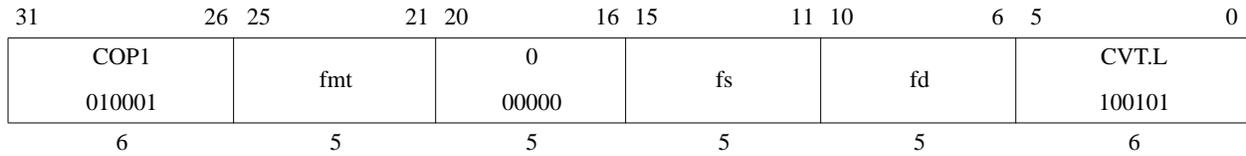
**Format:** CVT.L.S fd, fs **MIPS64 (MIPS III)**
CVT.L.D fd, fs **MIPS64 (MIPS III)**

**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 and round 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 $-2^{63}$ to $2^{63}$-1, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}-1$, is written to *fd*.

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *fmt* and *fd* for long fixed point—if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.
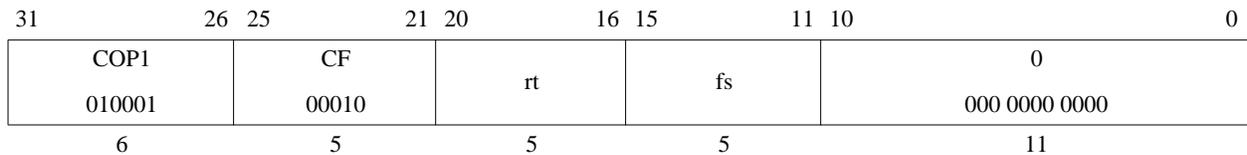
**Operation:**

    StoreFPR (fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact, Overflow

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| COP1 010001 | fmt 10000 | ft | fs | fd | CVT.PS 100110 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** CVT.PS.S fd, fs, ft      **MIPS64 (MIPS V)**

**Purpose:**

To convert two FP values to a paired single value

**Description:** $fd \leftarrow fs_{31..0} \,||\, ft_{31..0}$

The single-precision values in FPR *fs* and *ft* are written into FPR *fd* as a paired-single value. The value in FPR *fs* is written into the upper half, and the value in FPR *ft* is written into the lower half.
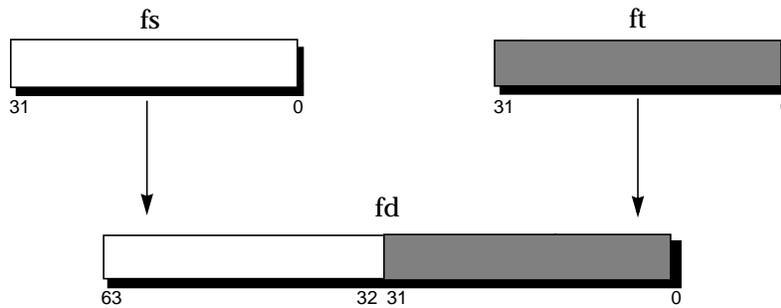


CVT.PS.S is similar to PLL.PS, except that it expects operands of format *S* instead of *PS*.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields *fs* and *ft* must specify FPRs valid for operands of type *S*; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *S*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
StoreFPR(fd, S, ValueFPR(fs,S) || ValueFPR(ft,S))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

*Floating Point Exceptions:*

Invalid Operation, Unimplemented Operation

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| COP1 010001 | | fmt | | 0 00000 | | fs | | fd | | CVT.S 100000 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:** CVT.S.D fd, fs                                   **MIPS32 (MIPS I)**
CVT.S.W fd, fs                                   **MIPS32 (MIPS I)**
CVT.S.L fd, fs                                   **MIPS64 (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 and rounded according to the current rounding mode in *FCSR*. The result is placed in FPR *fd*.

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *fmt* and *fd* for single floating point. If they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

**Operation:**

    StoreFPR(fd, S, ConvertFmt(ValueFPR(fs, fmt), fmt, S))

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact, Overflow, Underflow

## Floating Point Convert Pair Lower to Single Floating Point       CVT.S.PL

| 31       26 | 25       21 | 20       16 | 15       11 | 10       6 | 5       0 |
|---|---|---|---|---|---|
| COP1 | fmt | 0 | fs | fd | CVT.S.PL |
| 010001 | 10110 | 00000 | | | 101000 |
| 6 | 5 | 5 | 5 | 5 | 6 |
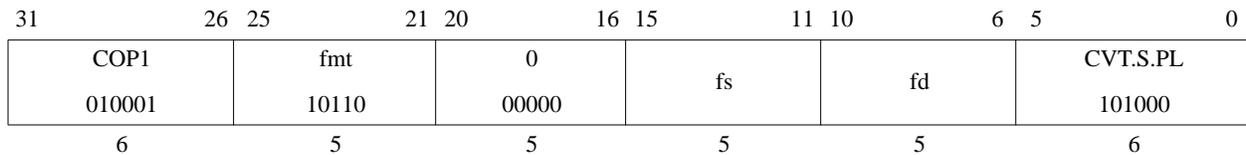
**Format:** CVT.S.PL fd, fs       **MIPS64 (MIPS V)**

**Purpose:**

To convert one half of a paired single FP value to single FP

**Description:** fd ← convert_and_round(fs)

The lower paired single value in FPR *fs*, in format *PS,* is converted to a value in single floating point format and rounded according to the current rounding mode in *FCSR*. The result is placed in FPR *fd*. This instruction can be used to isolate the lower half of a paired single value.

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *PS* and *fd* for single floating point. If they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *PS*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of CVT.S.PL is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
StoreFPR (fd, S, ConvertFmt(ValueFPR(fs, PS), PL, S))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact, Overflow, Underflow

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| COP1 | fmt | 0 | fs | fd | CVT.S.PU |
| 010001 | 10110 | 00000 | | | 100000 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**   CVT.S.PU fd, fs                                                                     **MIPS64 (MIPS V)**

**Purpose:**

To convert one half of a paired single FP value to single FP

**Description:** fd ← convert_and_round(fs)

The upper paired single value in FPR *fs*, in format *PS*, is converted to a value in single floating point format and rounded according to the current rounding mode in *FCSR*. The result is placed in FPR *fd*. This instruction can be used to isolate the upper half of a paired single value.

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *PS* and *fd* for single floating point. If they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *PS*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of CVT.S.PU is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
StoreFPR (fd, S, ConvertFmt(ValueFPR(fs, PS), PU, S))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact, Overflow, Underflow

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COP1 010001 | | fmt | | 0 00000 | | fs | | fd | | CVT.W 100100 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:**  CVT.W.S fd, fs  **MIPS32 (MIPS I)**
CVT.W.D fd, fs  **MIPS32 (MIPS I)**

**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 and 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, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *fd*.

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *fmt* and *fd* for word fixed point—if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

**Operation:**

    StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact, Overflow

## Doubleword Add                                                          DADD

| 31            26 | 25        21 | 20        16 | 15        11 | 10         6 | 5              0 |
|------------------|--------------|--------------|--------------|--------------|------------------|
| SPECIAL          | rs           | rt           | rd           | 0            | DADD             |
| 000000           |              |              |              | 00000        | 101100           |
| 6                | 5            | 5            | 5            | 5            | 6                |

**Format:** DADD rd, rs, rt                                    **MIPS64 (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:**

**Operation:**

```
temp ← (GPR[rs]_63||GPR[rs]) + (GPR[rt]_63||GPR[rt])
if (temp_64 ≠ temp_63) then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp_63..0
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<br>011000 | rs | rt | immediate |
| 6 | 5 | 5 | 16 |

**Format:** DADDI rt, rs, immediate                            **MIPS64 (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:**

**Operation:**

```
temp ← (GPR[rs]_63||GPR[rs]) + sign_extend(immediate)
if (temp_64 ≠ temp_63) then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp_63..0
endif
```

**Exceptions:**

Integer Overflow, Reserved Instruction

**Programming Notes:**

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

## Doubleword Add Immediate Unsigned

| 31          26 | 25      21 | 20    16 | 15                              0 |
|----------------|------------|----------|-----------------------------------|
| DADDIU<br>011001 | rs | rt | immediate |
| 6 | 5 | 5 | 16 |

**Format:** DADDIU rt, rs, immediate                           **MIPS64 (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:**

**Operation:**

    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 unsigned arithmetic such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

## Doubleword Add Unsigned

<div align="right">DADDU</div>

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| SPECIAL<br>000000 | rs | rt | rd | 0<br>00000 | DADDU<br>101101 |
| 6 | 5 | 5 | 5 | 5 | 6 |
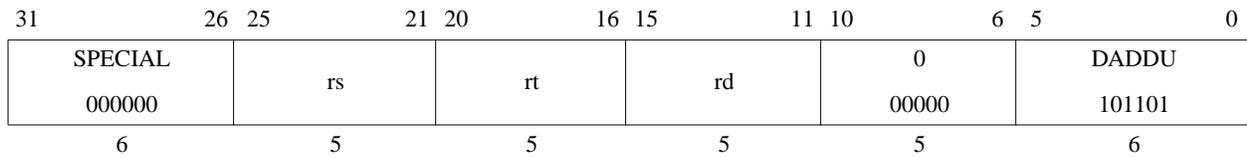
**Format:** DADDU   rd, rs, rt                                            **MIPS64 (MIPS III)**

**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:**

**Operation:**

```
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 unsigned arithmetic 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 |
|----------------|-------------|-------------|-------------|-------------|--------------|
| SPECIAL2       |             |             |             | 0           | DCLO         |
| 011100         | rs          | rt          | rd          | 00000       | 100101       |
| 6              | 5           | 5           | 5           | 5           | 6            |

**Format:**  DCLO rd, rs                                                          **MIPS64**

**Purpose:**

To count the number of leading ones in a doubleword

**Description:** rd ← count_leading_ones rs

The 64-bit word in GPR *rs* is scanned from most-significant to least-significant bit. The number of leading ones is counted and the result is written to GPR *rd*. If all 64 bits were set in GPR *rs*, the result written to GPR *rd* is 64.

**Restrictions:**

To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in  both the *rt* and *rd* fields of the instruction.  The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values.

**Operation:**

```
temp <- 64
for i in 63.. 0
    if GPR[rs]_i = 1 then
        temp <- 63 - i
        break
    endif
endfor
GPR[rd] <- temp
```

**Exceptions:**

None

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| SPECIAL2<br><br>011100 | rs | rt | rd | 0<br><br>00000 | DCLZ<br><br>100100 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**  DCLZ rd, rs **MIPS64**

**Purpose:**

To count the number of leading zeros in a doubleword

**Description:** rd ← count_leading_zeros rs

The 64-bit word in GPR *rs* is scanned from most significant to least significant bit. The number of leading zeros is counted and the result is written to GPR *rd*. If no bits were set in GPR *rs*, the result written to GPR *rd* is 64.

**Restrictions:**

To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the *rt* and *rd* fields of the instruction.  The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values.

**Operation:**

```
temp <- 64
for i in 63.. 0
    if GPR[rs]_i = 0 then
        temp <- 63 - i
        break
    endif
endfor
GPR[rd] <- temp
```

**Exceptions:**

None

## Doubleword Divide DDIV

| 31          26 | 25       21 | 20      16 | 15                        6 | 5         0 |
|----------------|-------------|------------|-----------------------------|-------------|
| SPECIAL<br>000000 | rs | rt | 0<br>00 0000 0000 | DDIV<br>011110 |
| 6 | 5 | 5 | 10 | 6 |
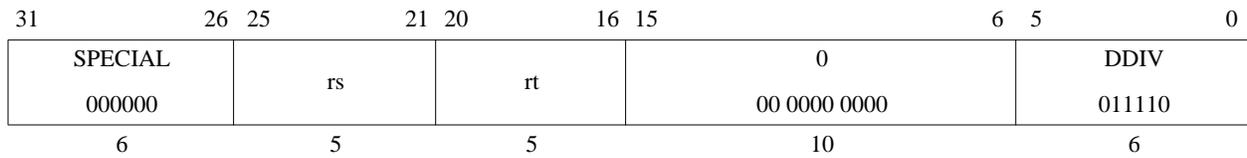
**Format:** DDIV   rs, rt                                                   **MIPS64 (MIPS III)**

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

**Operation:**

```
LO ← GPR[rs] div GPR[rt]
HI ← GPR[rs] mod GPR[rt]
```

**Exceptions:**

Reserved Instruction

**Programming Notes:**

See "Programming Notes" for the DIV instruction.

**Historical Perspective:**

In MIPS III, if either of the two instructions preceding the divide is an MFHI or MFLO, the result of the MFHI or MFLO is **UNPREDICTABLE**. Reads of the HI or LO special register must be separated from subsequent instructions that write to them by two or more instructions. This restriction was removed in MIPS IV and MIPS32 and all subsequent levels of the architecture.

| 31 26 | 25 21 | 20 16 | 15 6 | 5 0 |
|---|---|---|---|---|
| SPECIAL<br>000000 | rs | rt | 0<br>00 0000 0000 | DDIVU<br>011111 |
| 6 | 5 | 5 | 10 | 6 |
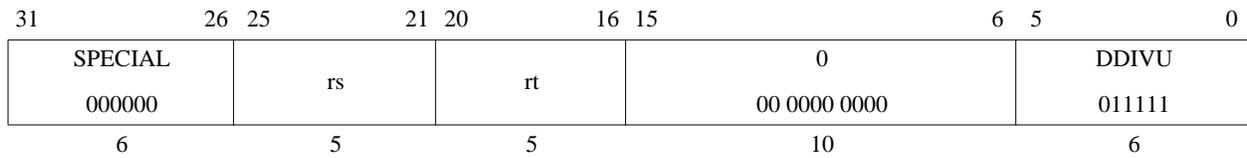
**Format:** DDIVU rs, rt **MIPS64 (MIPS III)**

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

**Operation:**

```
q ← (0 || GPR[rs]) div (0 || GPR[rt])
r ← (0 || GPR[rs]) mod (0 || GPR[rt])
LO ← q_{63..0}
HI ← r_{63..0}
```

**Exceptions:**

Reserved Instruction

**Programming Notes:**

See "Programming Notes" for the DIV instruction.

**Historical Perspective:**

In MIPS III, if either of the two instructions preceding the divide is an MFHI or MFLO, the result of the MFHI or MFLO is UNPREDICTABLE. Reads of the HI or LO special register must be separated from subsequent instructions that write to them by two or more instructions. This restriction was removed in MIPS IV and MIPS32 and all subsequent levels of the architecture.

| Debug Exception Return | | | | DERET |
|---|---|---|---|---|

| 31 | 26 | 25 | 24 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|
| COP0 | | CO | 0 | | DERET | |
| 010000 | | 1 | 000 0000 0000 0000 0000 | | 011111 | |
| 6 | | 1 | 19 | | 6 | |

**Format:** DERET                                                                                 EJTAG

**Purpose:**

To Return from a debug exception.

**Description:**

DERET returns from Debug Mode and resumes non-debug execution at the instruction whose address is contained in the *DEPC* register. DERET does not execute the next instruction (i.e. it has no delay slot).

**Restrictions:**

A DERET placed between an LL and SC instruction does not cause the SC to fail.

If the DEPC register with the return address for the DERET was modified by an MTC0 or a DMTC0 instruction, a CP0 hazard hazard exists that must be removed via software insertion of the appropriate number of SSNOP instructions.

The DERET instruction implements a software barrier for all changes in the CP0 state that could affect the fetch and decode of the instruction at the PC to which the DERET returns, such as changes to the effective ASID, user-mode state, and addressing mode.

This instruction is legal only if the processor is executing in Debug Mode.The operation of the processor is **UNDEFINED** if a DERET is executed in the delay slot of a branch or jump instruction.

**Operation:**

```
Debug_DM ← 0
Debug_IEXI ← 0
if IsMIPS16Implemented() then
    PC ← DEPC_63..1 ‖ 0
    ISAMode ← 0 ‖ DEPC_0
else
    PC ← DEPC
endif
```

**Exceptions:**

Coprocessor Unusable Exception
Reserved Instruction Exception

## Divide Word

**DIV**

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|
| SPECIAL<br>000000 | | rs | | rt | | 0<br>00 0000 0000 | | DIV<br>011010 | |
| 6 | | 5 | | 5 | | 10 | | 6 | |

**Format:** DIV rs, rt

**MIPS32 (MIPS I)**

**Purpose:**

To divide a 32-bit signed integers

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

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 sign-extended and placed into special register *LO* and the 32-bit remainder is sign-extended and placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

**Restrictions:**

If either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

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

**Operation:**

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
    UNPREDICTABLE
endif
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:**

No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions are 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 within 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 a zero is detected.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

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 interlocks 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.

**Historical Perspective:**

In MIPS 1 through MIPS III, if either of the two instructions preceding the divide is an MFHI or MFLO, the result of the MFHI or MFLO is UNPREDICTABLE. Reads of the HI or LO special register must be separated from subsequent instructions that write to them by two or more instructions. This restriction was removed in MIPS IV and MIPS32 and all subsequent levels of the architecture.

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|-------|-------|-------|-------|------|-----|
| COP1 010001 | fmt | ft | fs | fd | DIV 000011 |
| 6 | 5 | 5 | 5 | 5 | 6 |
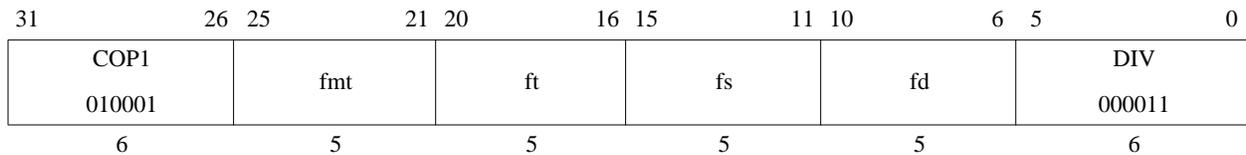
**Format:** DIV.S fd, fs, ft **MIPS32 (MIPS I)**
DIV.D fd, fs, ft **MIPS32 (MIPS I)**

**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*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

**Operation:**

```
StoreFPR (fd, fmt, ValueFPR(fs, fmt) / ValueFPR(ft, fmt))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Invalid Operation, Unimplemented Operation, Division-by-zero, Overflow, Underflow

| 31        26 | 25      21 | 20      16 | 15            6 | 5        0 |
|:---:|:---:|:---:|:---:|:---:|
| SPECIAL<br>000000 | rs | rt | 0<br>00 0000 0000 | DIVU<br>011011 |
| 6 | 5 | 5 | 10 | 6 |
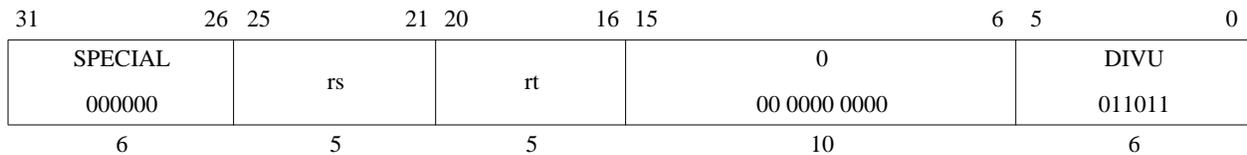
**Format:**   DIVU rs, rt                                     **MIPS32 (MIPS I)**

**Purpose:**

To divide a 32-bit unsigned integers

**Description:** $(LO, HI) \leftarrow$ 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 sign-extended and placed into special register *LO* and the 32-bit remainder is sign-extended and placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

**Restrictions:**

If either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

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
q   ← (0 || GPR[rs]31..0) div (0 || GPR[rt]31..0)
r   ← (0 || GPR[rs]31..0) mod (0 || GPR[rt]31..0)
LO  ← sign_extend(q31..0)
HI  ← sign_extend(r31..0)
```

**Exceptions:**

None

**Programming Notes:**

See "Programming Notes" for the DIV instruction.

**Historical Perspective:**

In MIPS 1 through MIPS III, if either of the two instructions preceding the divide is an MFHI or MFLO, the result of the MFHI or MFLO is UNPREDICTABLE. Reads of the HI or LO special register must be separated from subsequent instructions that write to them by two or more instructions. This restriction was removed in MIPS IV and MIPS32 and all subsequent levels of the architecture.

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 3 | 2 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COP0 010000 | | DMF 00001 | | rt | | rd | | 0 0000 0000 | | sel | |
| 6 | | 5 | | 5 | | 5 | | 8 | | 3 | |
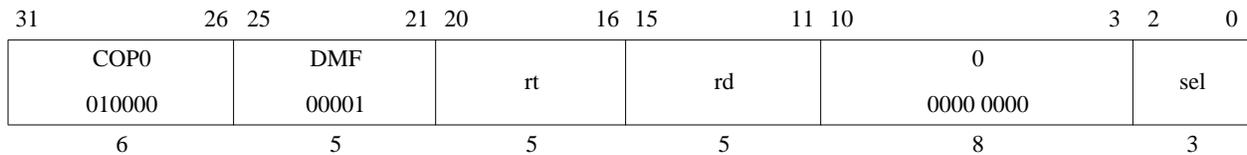
**Format:**  DMFC0 rt, rd                                                                     **MIPS64**
　　　　　　 DMFC0 rt, rd, sel                                                               **MIPS64**

**Purpose:**

To move the contents of a coprocessor 0 register to a general purpose register (GPR).

**Description:** rt ← CPR[0,rd,sel]

The contents of the coprocessor 0 register are loaded into GPR *rt*. Note that not all coprocessor 0 registers support the *sel* field. In those instances, the *sel* field must be zero.

**Restrictions:**

The results are **UNPREDICTABLE** if coprocessor 0 does not contain a register as specified by *rd* and *sel*, or if the coprocessor 0 register specified by *rd* and *sel* is a 32-bit register.

**Operation:**
```
    datadoubleword ← CPR[0,rd,sel]
    GPR[rt] ← datadoubleword
```

**Exceptions:**

Coprocessor Unusable

Reserved Instruction

## Doubleword Move from Floating Point

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| COP1 010001 | | DMF 00001 | | rt | | fs | | 0 000 0000 0000 | |
| 6 | | 5 | | 5 | | 5 | | 11 | |
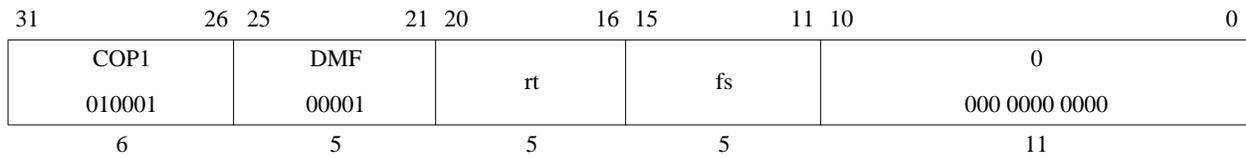
**Format:** DMFC1 rt,fs                                            **MIPS64 (MIPS III)**

**Purpose:**

To move a doubleword from an FPR to a GPR.

**Description:** rt← fs

The contents of FPR *fs* are loaded into GPR *rt*.

**Restrictions:**

**Operation:**
```
datadoubleword ← ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)
GPR[rt] ← datadoubleword
```

**Exceptions:**

Coprocessor Unusable

Reserved Instruction

**Historical Information:**

For MIPS III, the contents of GPR *rt* are undefined for the instruction immediately following DMFC1.

## Doubleword Move from Coprocessor 2 — DMFC2

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 3 | 2 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COP2 010010 | | DMF 00001 | | rt | | rd | | 0 000 0000 0 | | sel | |
| 6 | | 5 | | 5 | | 5 | | 8 | | 3 | |
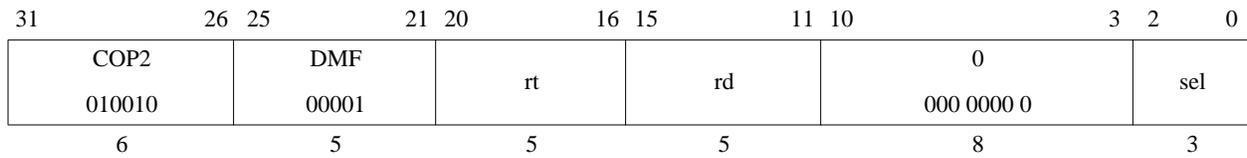
**Format:** DMFC2 rt, rd      **MIPS64**
        DMFC2, rt, rd,sel      **MIPS64**

**Purpose:**

To move a doubleword from a coprocessor 2 register to a GPR.

**Description:** rt ← CPR[2, rd, sel]

The contents of the coprocessor 2 register specified by the *rd* and *sel* fields are loaded into GPR *rt*. Note that not all coprocessor 2 registers may support the *sel* field. In those instances, the *sel* field must be zero.

**Restrictions:**

The results are **UNPREDICTABLE** if coprocessor 2 does not contain a register as specified by *rd* and *sel*, or if the coprocessor 2 register specified by *rd* and *sel* is a 32-bit register.

**Operation:**

```
datadoubleword ← CPR[2,rd,sel]
GPR[rt] ← datadoubleword
```

**Exceptions:**

Coprocessor Unusable

Reserved Instruction

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 3 | 2 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| COP0<br>010000 | | DMT<br>00101 | | rt | | rd | | 0<br>0000 0000 | | sel | |
| 6 | | 5 | | 5 | | 5 | | 8 | | 3 | |
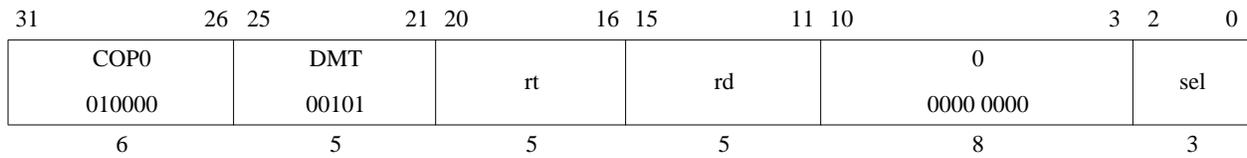
**Format:**  DMTC0 rt, rd                                                                    **MIPS64**
     DMTC0 rt, rd, sel                                                              **MIPS64**

**Purpose:**

To move a doubleword from a GPR to a coprocessor 0 register.

**Description:** CPR[0,rd,sel] ← rt

The contents of GPR *rt* are loaded into the coprocessor 0 register specified in the *rd* and *sel* fields. Note that not all coprocessor 0 registers support the *sel* field. In those instances, the *sel* field must be zero.

**Restrictions:**

The results are **UNPREDICTABLE** if coprocessor 0 does not contain a register as specified by *rd* and *sel*, or if the coprocessor 0 register specified by *rd* and *sel* is a 32-bit register.

**Operation:**

```
datadoubleword ← GPR[rt]
CPR[0,rd,sel] ← datadoubleword
```

**Exceptions:**

Coprocessor Unusable

Reserved Instruction

| 31          | 26 | 25          | 21 | 20   | 16 | 15   | 11 | 10                     | 0 |
|-------------|----|-------------|----|------|----|------|----|------------------------|---|
| COP1        |    | DMT         |    | rt   |    | fs   |    | 0                      |   |
| 010001      |    | 00101       |    |      |    |      |    | 000 0000 0000          |   |
| 6           |    | 5           |    | 5    |    | 5    |    | 11                     |   |
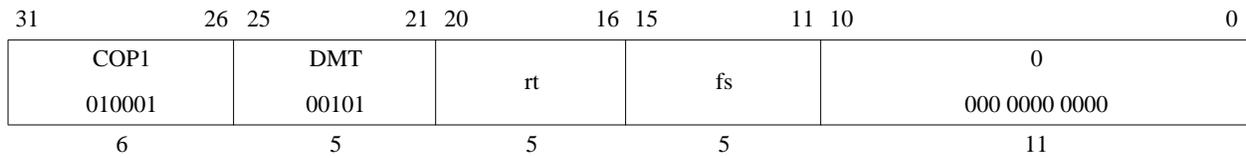
**Format:**  DMTC1 rt, fs                                                                **MIPS64 (MIPS III)**

**Purpose:**

To copy a doubleword from a GPR to an FPR

**Description:** fs← rt

The doubleword contents of FPR *fs* are placed into FPR *fs*.

**Restrictions:**

**Operation:**

```
datadoubleword ← GPR[rt]
StoreFPR(fs, UNINTERPRETED_DOUBLEWORD, datadoubleword)
```

**Exceptions:**

Coprocessor Unusable

Reserved Instruction

**Historical Information:**

For MIPS III, the contents of FPR *fs* are undefined for the instruction immediately following DMTC1.

| 31        26 | 25        21 | 20        16 | 15        11 | 10                    3 | 2        0 |
|--------------|--------------|--------------|--------------|-------------------------|------------|
| COP2<br>010010 | DMT<br>00101 | rt | rd | 0<br>0 0000 000 | sel |
| 6 | 5 | 5 | 5 | 8 | 3 |

**Format:**  DMTC2 rt,rd                                                                                   **MIPS64**
             DMTC2 rt, rd, sel                                                                          **MIPS64**

**Purpose:**

To move a doubleword from a GPR to a coprocessor 2 register.

**Description:** rd ← rt

The contents of GPR *rt* are loaded into the coprocessor 2 register specified by the *rd* and *sel* fields. Note that not all coprocessor 2 registers may support the *sel* field. In those instances, the *sel* field must be zero.

**Restrictions:**

The results are **UNPREDICTABLE** if coprocessor 2 does not contain a register as specified by *rd* and *sel*, or if the coprocessor 2 register specified by *rd* and *sel* is a 32-bit register.

**Operation:**

```
datadoubleword ← GPR[rt]
CPR[2,rd,sel]← datadoubleword
```

**Exceptions:**

Coprocessor Unusable

Reserved Instruction

| 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 **MIPS64 (MIPS III)**

**Purpose:**

To multiply 64-bit signed 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 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:**

**Operation:**

```
prod← GPR[rs] × GPR[rt]
LO  ← prod_{63..0}
HI  ← prod_{127..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 interlocks 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.

**Historical Perspective:**

In MIPS III, if either of the two instructions preceding the divide is an MFHI or MFLO, the result of the MFHI or MFLO is **UNPREDICTABLE**. Reads of the HI or LO special register must be separated from subsequent instructions that write to them by two or more instructions. This restriction was removed in MIPS IV and all subsequent levels of the architecture.

| 31      26 | 25      21 | 20      16 | 15                      6 | 5              0 |
|------------|------------|------------|---------------------------|------------------|
| SPECIAL    | rs         | rt         | 0                         | DMULTU           |
| 000000     |            |            | 00 0000 0000              | 011101           |
| 6          | 5          | 5          | 10                        | 6                |

**Format:**  DMULTU rs, rt                                                                   **MIPS64 (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:**

**Operation:**
```
prod← (0||GPR[rs]) × (0||GPR[rt])
LO  ← prod_63..0
HI  ← prod_127..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 interlocks 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.

**Historical Perspective:**

In MIPS III, if either of the two instructions preceding the divide is an MFHI or MFLO, the result of the MFHI or MFLO is UNPREDICTABLE. Reads of the HI or LO special register must be separated from subsequent instructions that write to them by two or more instructions. This restriction was removed in MIPS IV and all subsequent levels of the architecture.

## Doubleword Shift Left Logical

**DSLL**

| SPECIAL | 0 | rt | rd | sa | DSLL |
|---|---|---|---|---|---|
| 000000 | 00000 | | | | 111000 |
| 6 | 5 | 5 | 5 | 5 | 6 |

31       26 25      21 20      16 15      11 10      6 5      0
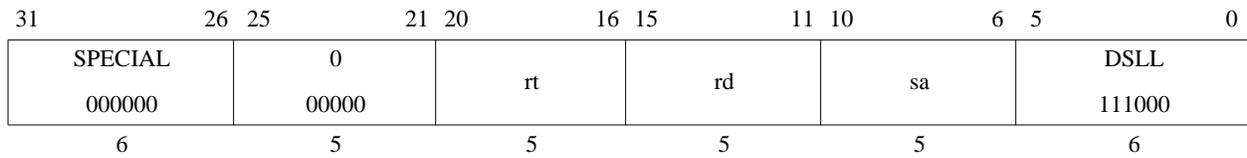
**Format:** DSLL rd, rt, sa

**MIPS64 (MIPS III)**

**Purpose:**

To execute a left-shift of 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 amount in the range 0 to 31 is specified by *sa*.

**Restrictions:**

**Operation:**

```
s     ← 0 || sa
GPR[rd]← GPR[rt](63-s)..0 || 0s
```

**Exceptions:**

Reserved Instruction

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| SPECIAL<br>000000 | 0<br>00000 | rt | rd | sa | DSLL32<br>111100 |
| 6 | 5 | 5 | 5 | 5 | 6 |
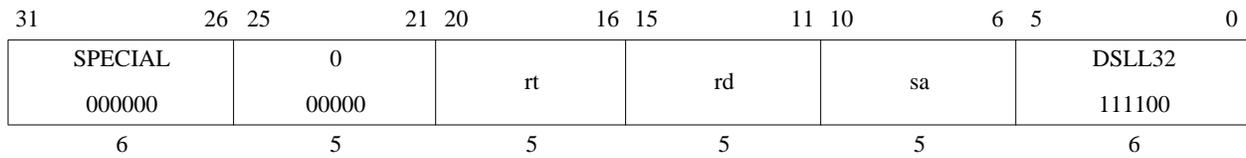
**Format:** DSLL32   rd, rt, sa **MIPS64 (MIPS III)**

**Purpose:**

To execute a left-shift of 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 amount in the range 0 to 31 is specified by *sa*.

**Restrictions:**

**Operation:**

```
s        ← 1 || sa    /* 32+sa */
GPR[rd]  ← GPR[rt]_(63-s)..0 || 0^s
```

**Exceptions:**

Reserved Instruction

## Doubleword Shift Left Logical Variable

**DSLLV**

| SPECIAL 000000 | rs | rt | rd | 0 00000 | DSLLV 010100 |
|:---:|:---:|:---:|:---:|:---:|:---:|
| 6 | 5 | 5 | 5 | 5 | 6 |

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

**Format:** DSLLV rd, rt, sa                **MIPS64 (MIPS III)**

**Purpose:**

To execute a left-shift of 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 amount in the range 0 to 63 is specified by the low-order 6 bits in GPR *rs*.

**Restrictions:**

**Operation:**

```
s          ← GPR[rs]₅..₀
GPR[rd]    ← GPR[rt]₍₆₃₋ₛ₎..₀ || 0ˢ
```

**Exceptions:**

Reserved Instruction

## Doubleword Shift Right Arithmetic                                      **DSRA**

| 31          26 | 25          21 | 20      16 | 15      11 | 10       6 | 5           0 |
|----------------|----------------|------------|------------|------------|---------------|
| SPECIAL        | 0              | rt         | rd         | sa         | DSRA          |
| 000000         | 00000          |            |            |            | 111011        |
| 6              | 5              | 5          | 5          | 5          | 6             |
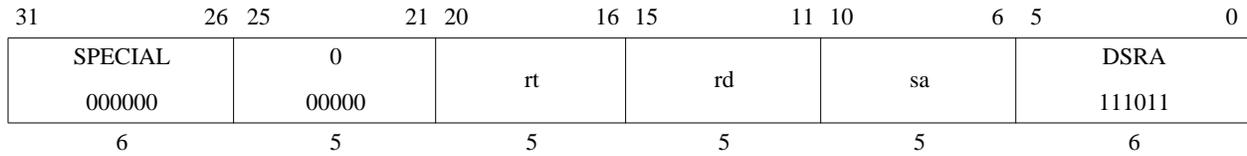
**Format:** DSRA   rd, rt, sa                                    **MIPS64 (MIPS III)**

**Purpose:**

To execute an arithmetic right-shift of 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 amount in the range 0 to 31 is specified by *sa.*

**Restrictions:**

**Operation:**

```
s          ← 0 || sa
GPR[rd]    ← (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<br>000000 | 0<br>00000 | rt | rd | sa | DSRA32<br>111111 |
| 6 | 5 | 5 | 5 | 5 | 6 |
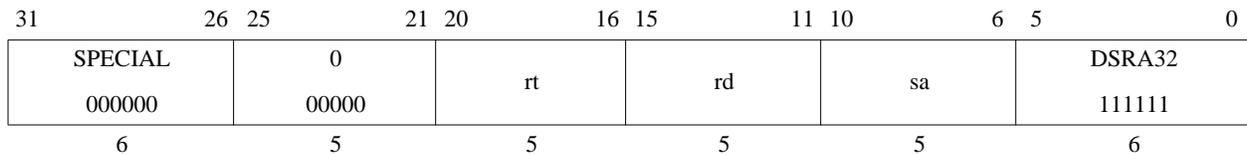
**Format:** DSRA32   rd, rt, sa

**MIPS64 (MIPS III)**

**Purpose:**

To execute an arithmetic right-shift of a doubleword by a fixed amount—32 to 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 amount in the range 32 to 63 is specified by *sa+32*.

**Restrictions:**

**Operation:**

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

$$s \leftarrow 1 \parallel sa \quad \text{/* } 32+sa \text{ */}$$
$$GPR[rd] \leftarrow (GPR[rt]_{63})^s \parallel GPR[rt]_{63..s}$$

**Exceptions:**

Reserved Instruction

## Doubleword Shift Right Arithmetic Variable

| 31          26 | 25        21 | 20        16 | 15        11 | 10          6 | 5           0 |
|----------------|--------------|--------------|--------------|---------------|---------------|
| SPECIAL        | rs           | rt           | rd           | 0             | DSRAV         |
| 000000         |              |              |              | 00000         | 010111        |
| 6              | 5            | 5            | 5            | 5             | 6             |

**Format:** DSRAV rd, rt, sa                                    **MIPS64 (MIPS III)**

**Purpose:**

To execute an arithmetic right-shift of 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 amount in the range 0 to 63 is specified by the low-order 6 bits in GPR *rs*.

**Restrictions:**

**Operation:**

```
s          ← GPR[rs]₅..₀
GPR[rd]    ← (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<br>000000 | 0<br>00000 | rt | rd | sa | DSRL<br>111010 |
| 6 | 5 | 5 | 5 | 5 | 6 |
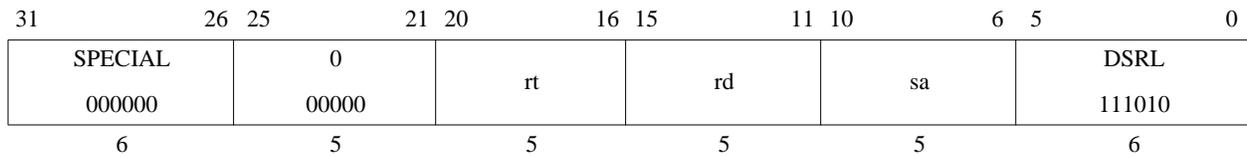
**Format:** DSRL    rd, rt, sa                                           **MIPS64 (MIPS III)**

**Purpose:**

To execute a logical right-shift of 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 amount in the range 0 to 31 is specified by *sa*.

**Restrictions:**

**Operation:**

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

**Exceptions:**

Reserved Instruction

## Doubleword Shift Right Logical Plus 32

<div align="right">

**DSRL32**

</div>

| 31         26 | 25         21 | 20      16 | 15      11 | 10      6 | 5          0 |
|---------------|---------------|------------|------------|-----------|--------------|
| SPECIAL       | 0             |            |            |           | DSRL32       |
| 000000        | 00000         | rt         | rd         | sa        | 111110       |
| 6             | 5             | 5          | 5          | 5         | 6            |
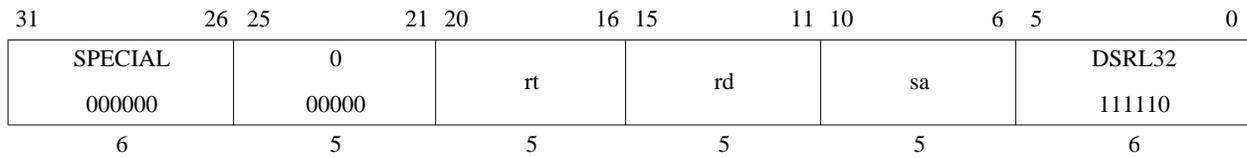
**Format:** DSRL32   rd, rt, sa

<div align="right">

**MIPS64 (MIPS III)**

</div>

**Purpose:**

To execute a logical right-shift of 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 amount in the range 32 to 63 is specified by *sa*+32.

**Restrictions:**

**Operation:**

```
s          ← 1 || sa    /* 32+sa */
GPR[rd]    ← 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<br>000000 | rs | rt | rd | 0<br>00000 | DSRLV<br>010110 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** DSRLV    rd, rt, rs                                                         **MIPS64 (MIPS III)**

**Purpose:**

To execute a logical right-shift of 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 amount in the range 0 to 63 is specified by the low-order 6 bits in GPR *rs*.

**Restrictions:**

**Operation:**

```
s          ← GPR[rs]5..0
GPR[rd]    ← 0s || GPR[rt]63..s
```

**Exceptions:**

Reserved Instruction

| 31          26 | 25          21 | 20          16 | 15          11 | 10          6 | 5          0 |
|----------------|----------------|----------------|----------------|---------------|--------------|
| SPECIAL<br>000000 | rs | rt | rd | 0<br>00000 | DSUB<br>101110 |
| 6 | 5 | 5 | 5 | 5 | 6 |
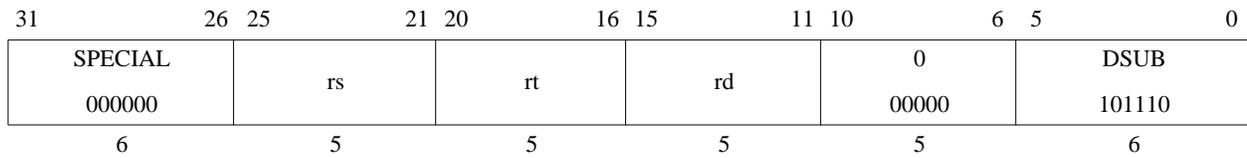
**Format:** DSUB   rd, rs, rt **MIPS64 (MIPS III)**

**Purpose:**

To subtract 64-bit integers; trap on 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:**

**Operation:**

```
temp ← (GPR[rs]₆₃||GPR[rs]) – (GPR[rt]₆₃||GPR[rt])
if (temp₆₄ ≠ temp₆₃) then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp₆₃..₀
endif
```

**Exceptions:**

Integer Overflow, Reserved Instruction

**Programming Notes:**

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

## Doubleword Subtract Unsigned

<div align="right">

**DSUBU**

</div>

| 31        26 | 25        21 | 20        16 | 15        11 | 10         6 | 5          0 |
|:---:|:---:|:---:|:---:|:---:|:---:|
| SPECIAL<br>000000 | rs | rt | rd | 0<br>00000 | DSUBU<br>101111 |
| 6 | 5 | 5 | 5 | 5 | 6 |
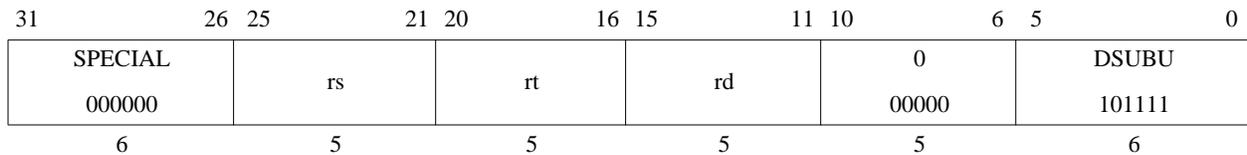
**Format:** DSUBU rd, rs, rt

<div align="right">

**MIPS64 (MIPS III)**

</div>

**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:**

**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 unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

| 31 | 26 | 25 | 24 | | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|
| COP0 | | CO | 0 | | | ERET | |
| 010000 | | 1 | 000 0000 0000 0000 0000 | | | 011000 | |
| 6 | | 1 | 19 | | | 6 | |
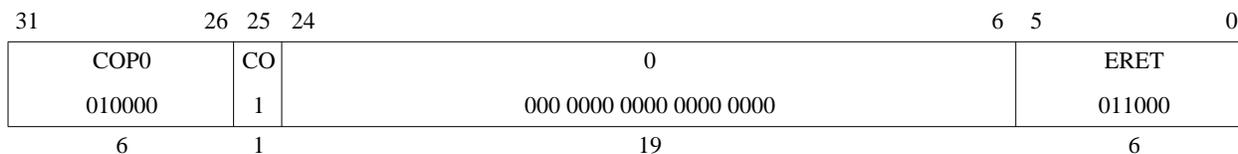
**Format:** ERET                                                                                                 MIPS32

**Purpose:**

To return from interrupt, exception, or error trap.

**Description:**

ERET returns to the interrupted instruction at the completion of interrupt, exception, or error trap processing. ERET does not execute the next instruction (i.e., it has no delay slot).

**Restrictions:**

The operation of the processor is **UNDEFINED** if an ERET is executed in the delay slot of a branch or jump instruction.

An ERET placed between an LL and SC instruction will always cause the SC to fail.

ERET implements a software barrier for all changes in the CP0 state that could affect the fetch and decode of the instruction at the PC to which the ERET returns, such as changes to the effective ASID, user-mode state, and addressing mode.

**Operation:**

```
if Status_ERL = 1 then
    temp ← ErrorEPC
    Status_ERL ← 0
else
    temp ← EPC
    Status_EXL ← 0
endif
if IsMIPS16Implemented() then
    PC ← temp_63..1 ∥ 0
    ISAMode ← temp_0
else
    PC ← temp
endif
LLbit ← 0
```

**Exceptions:**

      Coprocessor Unusable Exception

| 31          | 26 | 25      | 21 | 20        | 16 | 15   | 11 | 10  | 6 | 5               | 0 |
|-------------|----|---------|----|-----------|----|------|----|-----|---|-----------------|---|
| COP1        |    | fmt     |    | 0         |    | fs   |    | fd  |   | FLOOR.L         |   |
| 010001      |    |         |    | 00000     |    |      |    |     |   | 001011          |   |
| 6           |    | 5       |    | 5         |    | 5    |    | 5   |   | 6               |   |

**Format:**   FLOOR.L.S fd, fs                                   **MIPS64 (MIPS III)**
              FLOOR.L.D fd, fs                                   **MIPS64 (MIPS III)**

**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 and rounded 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, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation Enable bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}$–1, is written to *fd*.

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *fmt* and *fd* for long fixed point—if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact, Overflow

## Floating Point Floor Convert to Word Fixed Point

**FLOOR.W.fmt**

| 31        26 | 25        21 | 20        16 | 15        11 | 10        6 | 5        0 |
|:---:|:---:|:---:|:---:|:---:|:---:|
| COP1<br>010001 | fmt | 0<br>00000 | fs | fd | FLOOR.W<br>001111 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**  FLOOR.W.S  fd, fs  **MIPS32 (MIPS II)**
 FLOOR.W.D  fd, fs  **MIPS32 (MIPS II)**

**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 and rounded 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, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *fd*.

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *fmt* and *fd* for word fixed point—if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

**Operation:**

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact, Overflow

| Jump | J |
|------|---|

| 31 | 26 | 25 | 0 |
|----|----|----|---|
| J<br>000010 | | instr_index | |
| 6 | | 26 | |

**Format:** J target                                                                                          **MIPS32 (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 that follows the jump, in the branch delay slot, before executing the jump itself.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:
I+1:PC ← PC_GPRLEN..28 || instr_index || 0²
```

**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 from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the jump instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.

```
 31            26  25                                                                    0
┌─────────────────┬──────────────────────────────────────────────────────────────────────┐
│      JAL        │                                                                        │
│                 │                              instr_index                               │
│     000011      │                                                                        │
└─────────────────┴──────────────────────────────────────────────────────────────────────┘
        6                                          26
```
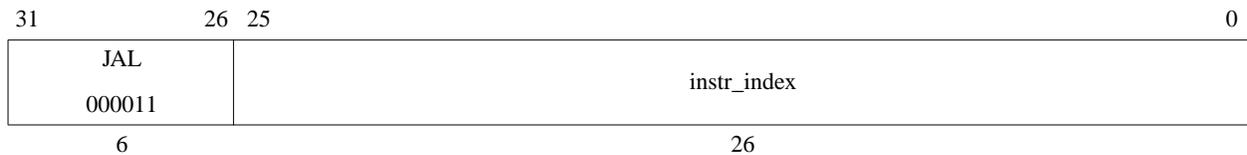
**Format:**   `JAL target`                                                              **MIPS32 (MIPS I)**

**Purpose:**

To execute a 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, at which location execution continues 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 that follows the jump, in the branch delay slot, before executing the jump itself.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:   GPR[31]← PC + 8
I+1: PC     ← PC_GPRLEN..28 || instr_index || 0²
```

**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 from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the branch instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SPECIAL 000000 | | rs | | 0 00000 | | rd | | hint | | JALR 001001 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:** JALR rs (rd = 31 implied)                                  **MIPS32 (MIPS I)**
         JALR rd, rs                                              **MIPS32 (MIPS I)**

**Purpose:**

To execute a 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 continues after a procedure call.

*For processors that do not implement the MIPS16 ASE:*

- Jump to the effective target address in GPR *rs*. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

*For processors that do implement the MIPS16 ASE:*

- Jump to the effective target address in GPR *rs*. Set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

At this time the only defined hint field value is 0, which sets default handling of JALR. Future versions of the architecture may define additional hint values.

**Restrictions:**

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

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16 ASE, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction. For processors that do implement the MIPS16 ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:   temp ← GPR[rs]
     GPR[rd] ← PC + 8
I+1: if Config1_CA = 0 then
         PC ← temp
     else
         PC ← temp_GPRLEN-1..1 || 0
         ISAMode ← temp_0
     endif
```

**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.

| 31 | 26 | 25 | 21 | 20 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|
| SPECIAL<br>000000 | | rs | | 0<br>00 0000 0000 | | hint | | JR<br>001000 | |
| 6 | | 5 | | 10 | | 5 | | 6 | |
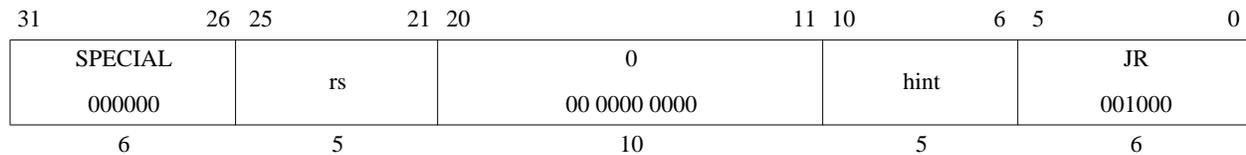
**Format:** JR rs                                                                     **MIPS32 (MIPS I)**

**Purpose:**

To execute a 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.

For processors that implement the MIPS16 ASE, set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

**Restrictions:**

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16 ASE, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction. For processors that do implement the MIPS16 ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

At this time the only defined hint field value is 0, which sets default handling of JR. Future versions of the architecture may define additional hint values.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:   temp ← GPR[rs]
I+1: if Config1_CA = 0 then
        PC ← temp
     else
        PC ← temp_{GPRLEN-1..1} || 0
        ISAMode ← temp_0
     endif
```

**Exceptions:**

None

**Programming Notes:**

Software should use the value 31 for the *rs* field of the instruction word on return from a JAL, JALR, or BGEZAL, and should use a value other than 31 for remaining uses of JR.

| 31            | 26 | 25      | 21 | 20    | 16 | 15          0 |
|---------------|----|---------|----|-------|----|----------------|
| LB<br>100000  |    | base    |    | rt    |    | offset         |
| 6             |    | 5       |    | 5     |    | 16             |
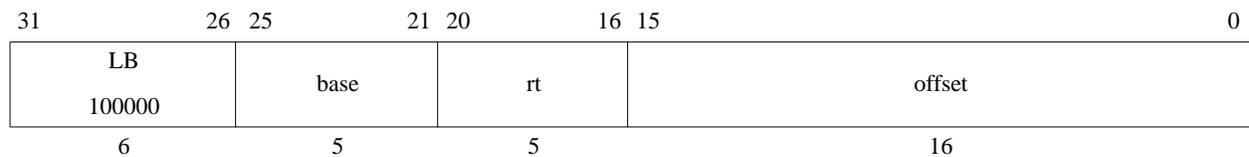
**Format:**   LB rt, offset(base)                                                            **MIPS32 (MIPS I)**

**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:**

```
vAddr     ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr     ← pAddr_{PSIZE-1..3} || (pAddr_{2..0} xor ReverseEndian^3)
memdoubleword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte      ← vAddr_{2..0} xor BigEndianCPU^3
GPR[rt]← sign_extend(memdoubleword_{7+8*byte..8*byte})
```

**Exceptions:**

TLB Refill, TLB Invalid, Address Error

## Load Byte Unsigned <span style="float:right">LBU</span>

| 31 26 | 25 21 | 20 16 | 15 0 |
|---|---|---|---|
| LBU<br>100100 | base | rt | offset |
| 6 | 5 | 5 | 16 |

**Format:** LBU rt, offset(base) <span style="float:right">**MIPS32 (MIPS I)**</span>

**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:**

```
vAddr    ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr    ← pAddr_{PSIZE-1..3} || (pAddr_{2..0} xor ReverseEndian^3)
memdoubleword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte     ← vAddr_{2..0} xor BigEndianCPU^3
GPR[rt]← zero_extend(memdoubleword_{7+8*byte..8*byte})
```

**Exceptions:**

TLB Refill, TLB Invalid, Address Error

**Load Doubleword**                                                                                           **LD**

| 31        26 | 25        21 | 20      16 | 15                                          0 |
|:---:|:---:|:---:|:---:|
| LD<br>110111 | base | rt | offset |
| 6 | 5 | 5 | 16 |
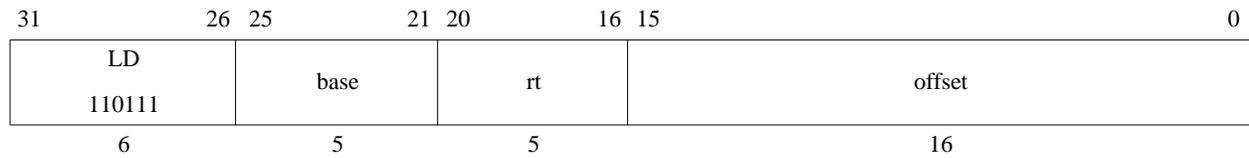
**Format:** `LD rt, offset(base)`                                                          **MIPS64 (MIPS III)**

**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 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

```
vAddr        ← sign_extend(offset) + GPR[base]
if vAddr₂..₀ ≠ 0³ then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[rt]      ← memdoubleword
```

**Exceptions:**

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

## Load Doubleword to Floating Point

**LDC1**

| 31            26 | 25            21 | 20        16 | 15                       0 |
|:---:|:---:|:---:|:---:|
| LDC1<br>110101 | base | ft | offset |
| 6 | 5 | 5 | 16 |

**Format:** LDC1 ft, offset(base)                                        **MIPS32 (MIPS II)**

**Purpose:**

To load a doubleword from memory to an FPR

**Description:** ft $\leftarrow$ 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.

**Restrictions:**

An Address Error exception occurs if EffectiveAddress$_{2..0} \neq 0$ (not doubleword-aligned).

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 0³ then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
StoreFPR(ft, UNINTERPRETED_DOUBLEWORD, memdoubleword)
```

**Exceptions:**

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

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|----|----|----|----|----|----|----|----|
| LDC2<br>110110 | | base | | rt | | offset | |
| 6 | | 5 | | 5 | | 16 | |

**Format:** LDC2 rt, offset(base) **MIPS32**

**Purpose:**

To load a doubleword from memory to a Coprocessor 2 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 placed in Coprocessor 2 register *rt*. 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 EffectiveAddress$_{2..0}$ ≠ 0 (not doubleword-aligned).

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then SignalException(AddressError) endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
CPR[2,rt,0] ← memdoubleword
```

**Exceptions:**

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

| 31           26 | 25           21 | 20        16 | 15                                              0 |
|-----------------|-----------------|--------------|---------------------------------------------------|
| LDL<br>011010   | base            | rt           | offset                                            |
| 6               | 5               | 5            | 16                                                |

**Format:**   `LDL rt, offset(base)`                                                           **MIPS64 (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 8 consecutive bytes forming a doubleword *(DW)* in memory, starting at an arbitrary byte boundary.

A part of *DW*, the most-significant 1 to 8 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.

**Figure 3-3 Unaligned Doubleword Load Using LDL and LDR**
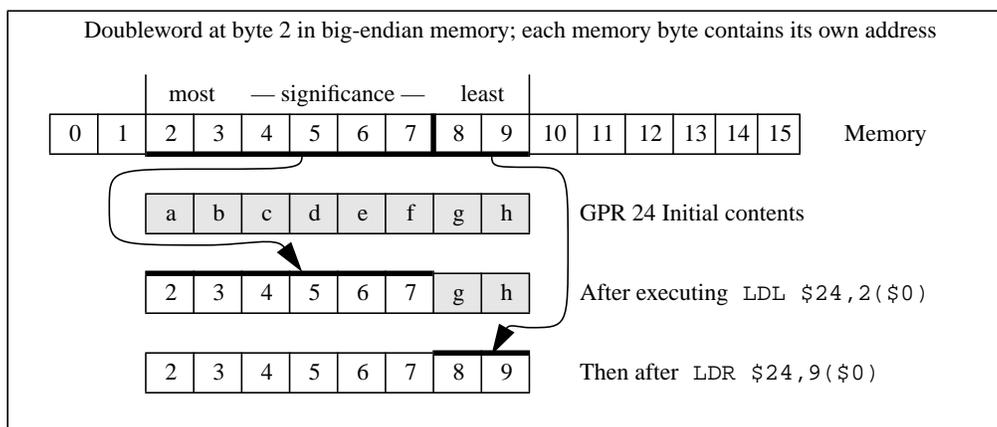


Figure 3-3 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 6 bytes, is located in the aligned doubleword starting with the most-significant byte at 2. LDL first loads these 6 bytes into the left part of the destination register and leaves the remainder of the destination unchanged. The complementary LDR next loads the remainder of the unaligned doubleword.

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned doubleword—the low 3 bits of the address (*vAddr2..0*)—and the current byte-ordering mode of the processor (big- or little-endian). Figure 3-4 shows the bytes loaded for every combination of offset and byte ordering.

**Figure 3-4 Bytes Loaded by LDL Instruction**
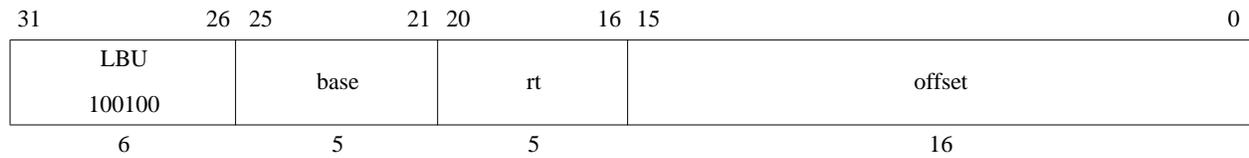


**Restrictions:**

**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← 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
memdoubleword← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
GPR[rt]← memdoublworde_{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<br>011011    | base         | rt         | offset                         |
| 6                | 5            | 5          | 16                             |
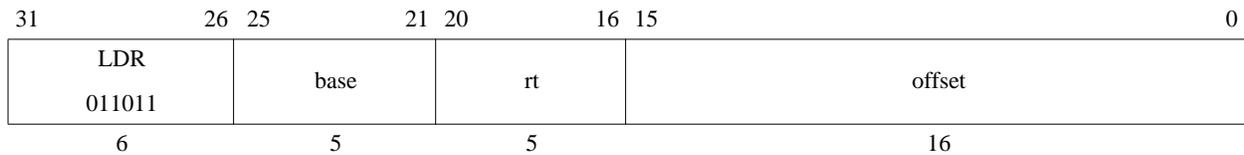
**Format:**  `LDR rt, offset(base)`                                   **MIPS64 (MIPS III)**

**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 8 consecutive bytes forming a doubleword *(DW)* in memory, starting at an arbitrary byte boundary.

A part of *DW*, the least-significant 1 to 8 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.

Figure 3-5 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. Two bytes of the *DW* are located in the aligned doubleword containing the least-significant byte at 9. LDR first loads these 2 bytes into the right part of the destination register, and leaves the remainder of the destination unchanged. The complementary LDL next loads the remainder of the unaligned doubleword.

**Figure 3-5 Unaligned Doubleword Load Using LDR and LDL**
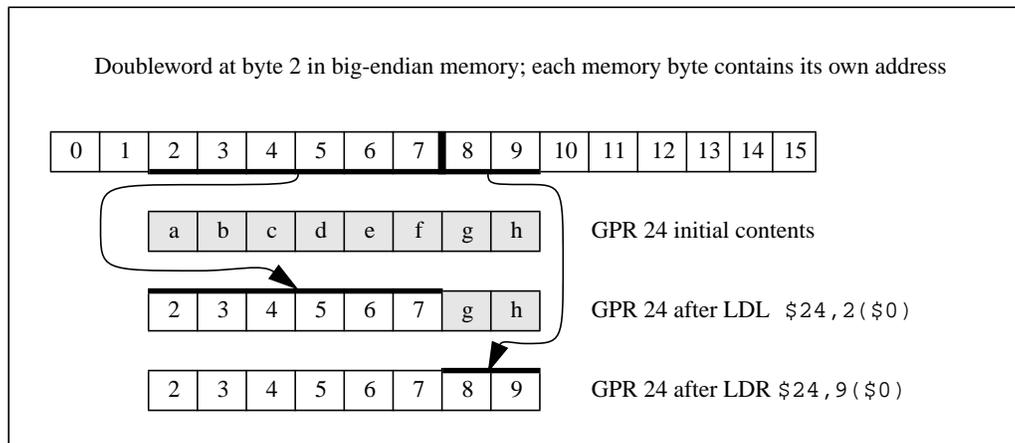
The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned doubleword—the low 3 bits of the address (*vAddr2..0)*—and the current byte-ordering mode of the processor (big- or little-endian).

Figure 3-6 shows the bytes loaded for every combination of offset and byte ordering.

**Figure 3-6 Bytes Loaded by LDR Instruction**



**Restrictions:**

**Operation: 64-bit processors**

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← 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_{2..0} xor BigEndianCPU^3
memdoubleword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
GPR[rt] ← GPR[rt]_{63..64-8*byte} || memdoubleword_{63..8*byte}
```

**Exceptions:**

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

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| COP1X<br>010011 | | base | | index | | 0<br>00000 | | fd | | LDXC1<br>000001 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |
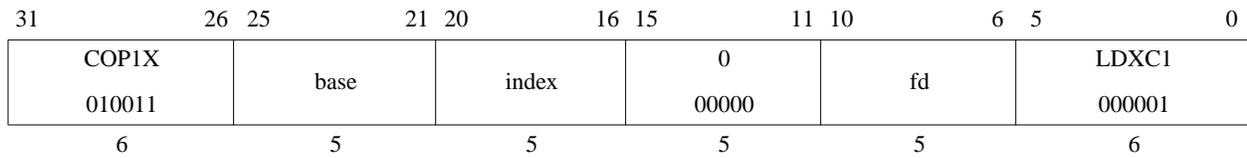
**Format:** `LDXC1 fd, index(base)` **MIPS64 (MIPS IV)**

**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.
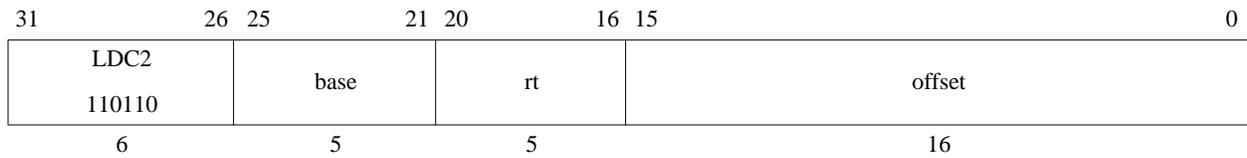
**Restrictions:**

An Address Error exception occurs if EffectiveAddress$_{2..0}$ ≠ 0 (not doubleword-aligned).

**Operation:**

```
vAddr ← GPR[base] + GPR[index]
if vAddr2..0 ≠ 0³ then SignalException(AddressError) endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
StoreFPR (fd, UNINTERPRETED_DOUBLEWORD, memdoubleword)
```

**Exceptions:**

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

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|
| LH<br>100001 | | base | | rt | | offset | |
| 6 | | 5 | | 5 | | 16 | |
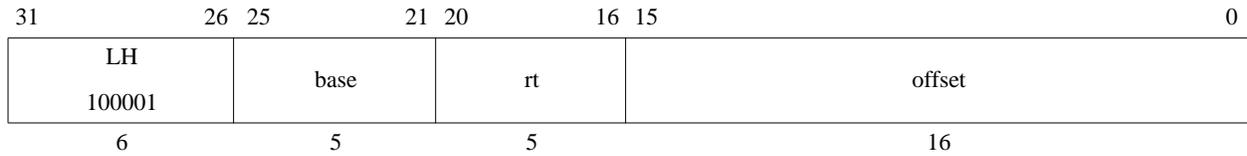
**Format:** LH rt, offset(base)                                                                        **MIPS32 (MIPS I)**

**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.
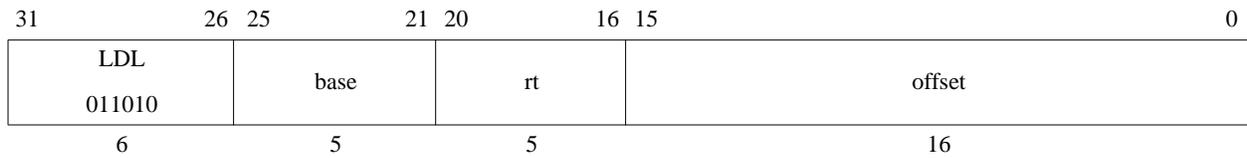
**Restrictions:**

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

**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr₀ ≠ 0 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr₂..₀ xor (ReverseEndian² || 0))
memdoubleword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte   ← vAddr₂..₀ xor (BigEndianCPU² || 0)
GPR[rt] ← sign_extend(memdoubleword₁₅₊₈*byte..₈*byte)
```

**Exceptions:**

TLB Refill, TLB Invalid, Bus Error, Address Error

| 31 | | 26 | 25 | | 21 | 20 | | 16 | 15 | | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| LHU 100101 | | | base | | | rt | | | offset | | |
| 6 | | | 5 | | | 5 | | | 16 | | |

**Format:** `LHU rt, offset(base)` **MIPS32 (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.

**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr₀ ≠ 0 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE−1..3 || (pAddr₂..₀ xor (ReverseEndian² || 0))
memdoubleword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte   ← vAddr₂..₀ xor (BigEndianCPU² || 0)
GPR[rt] ← zero_extend(memdoubleword₁₅₊₈*byte..8*byte)
```

**Exceptions:**

TLB Refill, TLB Invalid, Address Error

## Load Linked Word    LL

```
31        26 25        21 20      16 15                          0
┌──────────┬───────────┬──────────┬─────────────────────────────┐
│    LL    │           │          │                             │
│          │   base    │    rt    │           offset            │
│  110000  │           │          │                             │
└──────────┴───────────┴──────────┴─────────────────────────────┘
      6          5          5                   16
```

**Format:**  `LL rt, offset(base)`    **MIPS32 (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 the 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 can be only one active RMW sequence per processor.

When an LL is executed it starts an 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.

Executing LL on one processor does not cause an action that, by itself, causes 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.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr₁..₀ ≠ 0² then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr₂..₀ xor (ReverseEndian || 0²))
memdoubleword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr₂..₀ xor (BigEndianCPU || 0²)
GPR[rt] ← sign_extend(memdoubleword₃₁₊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.

| 31 26 | 25 21 | 20 16 | 15 0 |
|---|---|---|---|
| LLD<br>110100 | base | rt | offset |
| 6 | 5 | 5 | 16 |
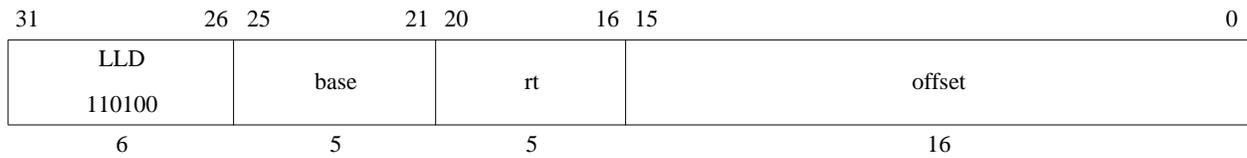
**Format:** `LLD   rt, offset(base)`                                                         **MIPS64 (MIPS III)**

**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 can be only one active RMW sequence per processor.

When an LLD is executed it starts the active RMW sequence and replaces 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 complete and fails.

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.

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

**Operation:**

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

**Exceptions:**

TLB Refill, TLB Invalid, Address Error, Reserved Instruction

## Load Upper Immediate                                                                          **LUI**

```
 31           26  25            21  20         16  15                                    0
┌────────────┬────────────┬──────────────┬────────────────────────────────────┐
│    LUI     │     0      │              │                                    │
│            │            │      rt      │             immediate              │
│   001111   │   00000    │              │                                    │
└────────────┴────────────┴──────────────┴────────────────────────────────────┘
      6             5             5                         16
```
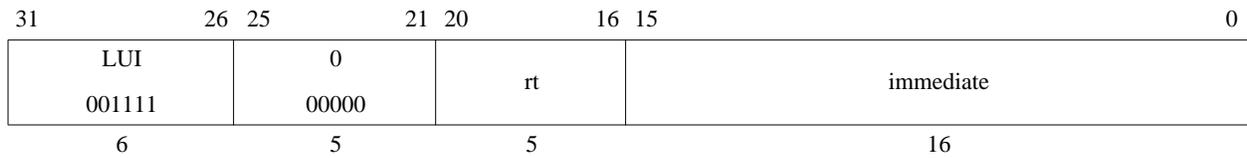
**Format:** `LUI rt, immediate`                                                    **MIPS32 (MIPS I)**

**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

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| COP1X 010011 | | base | | index | | 0 00000 | | fd | | LUXC1 000101 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |
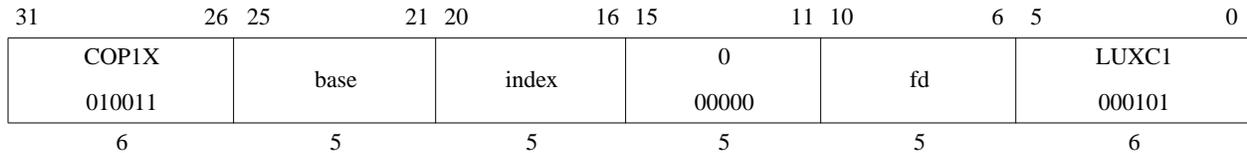
**Format:** `LUXC1 fd, index(base)` **MIPS64 (MIPS V)**

**Purpose:**

To load a doubleword from memory to an FPR (GPR+GPR addressing), ignoring alignment

**Description:** fd ← memory[(base+index)$_{PSIZE-1..3}$]

The contents of the 64-bit doubleword at the memory location specified by the 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. The effective address is doubleword-aligned; EffectiveAddress$_{2..0}$ are ignored.

**Restrictions:**

The result of this instruction is undefined if the processor is executing in 16 FP registers mode.

**Operation:**

```
vAddr  ← (GPR[base]+GPR[index])₆₃..₃ || 0³
(pAddr, CCA) ← AddressTranslation(vaddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
StoreFPR(ft, UNINTERPRETED, memdoubleword)
```

**Exceptions:**

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

**Load Word**                                                                                           **LW**

| 31          26 | 25          21 | 20          16 | 15                                    0 |
|----------------|----------------|----------------|------------------------------------------|
| LW<br>100011   | base           | rt             | offset                                   |
| 6              | 5              | 5              | 16                                       |
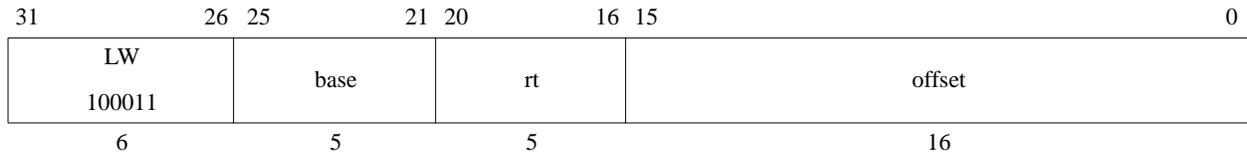
**Format:** `LW rt, offset(base)`                                                             **MIPS32 (MIPS I)**

**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 2 least-significant bits of the address is non-zero, an Address Error exception occurs.
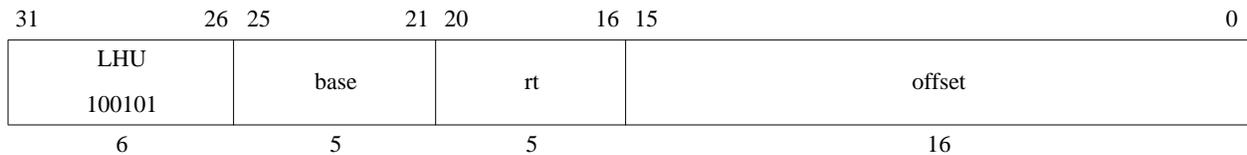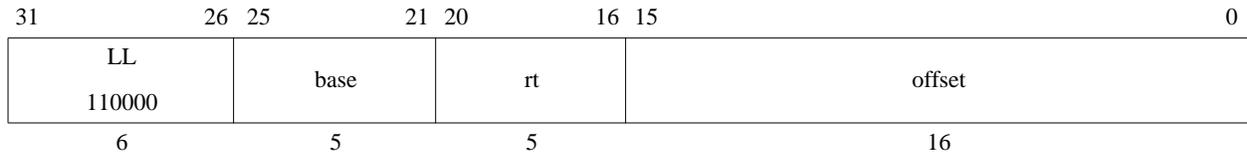
**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr₁..₀ ≠ 0² then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr₂..₀ xor (ReverseEndian || 0²))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr₂..₀ xor (BigEndianCPU || 0²)
GPR[rt]← sign_extend(memdoubleword₃₁₊₈*byte..₈*byte)
```

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

**Exceptions:**

TLB Refill, TLB Invalid, Bus Error, Address Error

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|
| LWC1 110001 | | base | | rt | | offset | |
| 6 | | 5 | | 5 | | 16 | |

**Format:** LWC1 ft, offset(base) **MIPS32 (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.

**Restrictions:**

An Address Error exception occurs if EffectiveAddress$_{1..0} \neq 0$ (not word-aligned).

**Operation:**

```
/* mem is aligned 64 bits from memory.  Pick out correct bytes. */
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
StoreFPR(ft, UNINTERPRETED_WORD,
         sign_extend(memdoubleword31+8*bytesel..8*bytesel))
```

**Exceptions:**

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

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|----|----|----|----|----|----|----|----|
| LWC2<br>110010 | | base | | rt | | offset | |
| 6 | | 5 | | 5 | | 16 | |
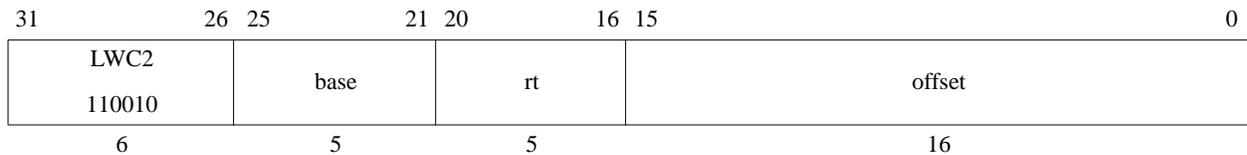
**Format:** `LWC2 rt, offset(base)`                                                                **MIPS32 (MIPS I)**

**Purpose:**

To load a word from memory to a COP2 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 placed into the low word of COP2 (Coprocessor 2) general register *rt*. 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 EffectiveAddress$_{1..0} \neq 0$ (not word-aligned).

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr12..0 ≠ 0² then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0²))
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
bytesel ← vAddr2..0 xor (BigEndianCPU || 0²)
CPR[2,rt,0] ← sign_extend(memdoubleword31+8*bytesel..8*bytesel)
```

**Exceptions:**

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

| 31          26 | 25          21 | 20     16 | 15                              0 |
|----------------|----------------|-----------|----------------------------------|
| LWL<br>100010  | base           | rt        | offset                           |
| 6              | 5              | 5         | 16                               |
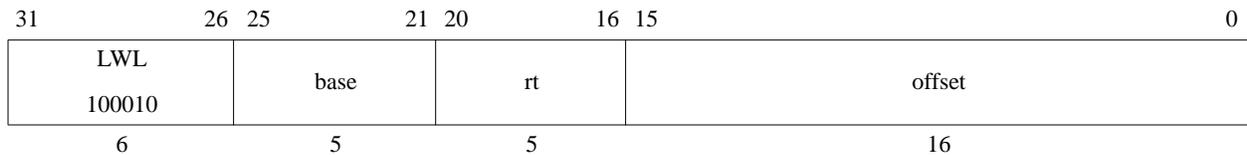
**Format:** `LWL rt, offset(base)`                                           **MIPS32 (MIPS I)**

**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 4 consecutive bytes forming a word *(W)* in memory starting at an arbitrary byte boundary.

The most-significant 1 to 4 bytes of *W* is in the aligned word containing the *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.

For 64-bit GPR *rt* registers, 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.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2 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

**Figure 3-7 Unaligned Word Load Using LWL and LWR**
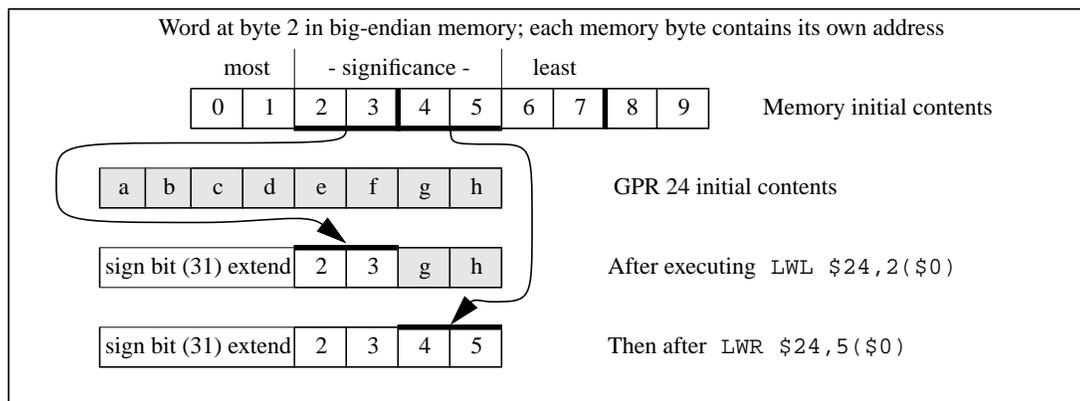
The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address (vAddr$_{1..0}$), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.

**Figure 3-8 Bytes Loaded by LWL Instruction**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Memory contents and byte offsets | | | | Initial contents of Dest Register | | | |

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

| a | b | c | d | e | f | g | h |

most        — significance —        least

Destination 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.

**Restrictions:**

None

**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_{PSIZE-1..3} || (pAddr_{2..0} xor ReverseEndian³)
if BigEndianMem = 0 then
    pAddr← pAddr_{PSIZE-1..3} || 0³
endif
byte   ← 0 || (vAddr_{1..0} xor BigEndianCPU²)
word   ← vAddr_2 xor BigEndianCPU
memdoubleword← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp   ← memdoubleword_{31+32*word-8*byte..32*word} || GPR[rt]_{23-8*byte..0}
GPR[rt]← (temp_{31})³² || temp
```

**Exceptions:**

None

TLB Refill, TLB Invalid, Bus Error, Address Error

**Programming Notes:**

The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

**Historical Information**

In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.

| 31          26 | 25        21 | 20      16 | 15                                        0 |
|----------------|--------------|------------|---------------------------------------------|
| LWR<br><br>100110 | base | rt | offset |
| 6 | 5 | 5 | 16 |

**Format:**  `LWR rt, offset(base)`                                        **MIPS32 (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 4 consecutive bytes forming a word *(W)* in memory starting at an arbitrary byte boundary.

A part of *W*, the least-significant 1 to 4 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 (that is, when all 4 bytes are loaded), then the new sign bit value is copied into bits 63..32. If bit 31 is not loaded, 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 a sign-extended word value in the destination register.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the least-significant byte at 5. First, LWR loads these 2 bytes into the right part of the destination register. Next, the complementary LWL loads the remainder of the unaligned word.

**Figure 3-9 Unaligned Word Load Using LWL and LWR**
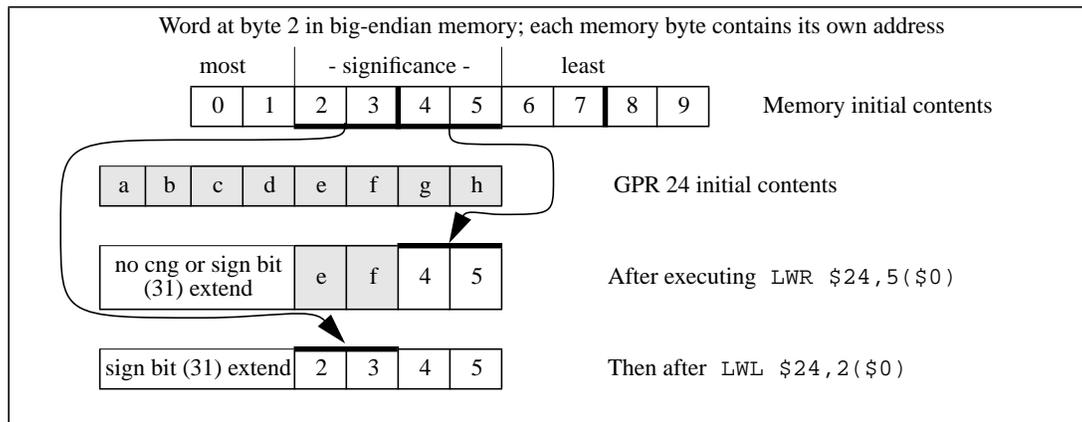


The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address ($vAddr_{1..0}$), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.

**Figure 3-10 Bytes Loaded by LWL Instruction**

| Memory contents and byte offsets | | | | | Initial contents of Dest Register | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | ←big-endian | | | | | | | | |
| I | J | K | L | offset (vAddr$_{1..0}$) | a | b | c | d | e | f | g | h |
| 3 | 2 | 1 | 0 | ←little-endian | most | | — significance — | | | | | least |
| most | | least | | | | | | | | | | |
| — significance — | | | | | | | | | | | | |

| 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 |

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

**Restrictions:**

None

**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← 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_1..0 xor BigEndianCPU^2
word   ← vAddr_2 xor BigEndianCPU
memdoubleword← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp   ← GPR[rt]_31..32-8*byte || memdoubleword_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, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

**Historical Information**

In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.

| 31        26 | 25       21 | 20     16 | 15                                    0 |
|:---:|:---:|:---:|:---:|
| LWU  100111 | base | rt | offset |
| 6 | 5 | 5 | 16 |

**Format:** `LWU rt, offset(base)`                                                            **MIPS64 (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 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr₁..₀ ≠ 0² then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr₂..₀ xor (ReverseEndian || 0²))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr₂..₀ xor (BigEndianCPU || 0²)
GPR[rt]← 0³² || memdoubleword₃₁+₈*byte..₈*byte
```

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

**Exceptions:**

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

| 31          | 26 | 25     | 21 | 20    | 16 | 15        | 11 | 10 | 6 | 5          | 0 |
|-------------|----|--------|----|-------|----|-----------|----|----|---|------------|---|
| COP1X       |    | base   |    | index |    | 0         |    | fd |   | LWXC1      |   |
| 010011      |    |        |    |       |    | 00000     |    |    |   | 000000     |   |
| 6           |    | 5      |    | 5     |    | 5         |    | 5  |   | 6          |   |

**Format:**  `LWXC1 fd, index(base)`                                                        **MIPS64 (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.

**Restrictions:**

An Address Error exception occurs if EffectiveAddress$_{1..0} \neq 0$ (not word-aligned).

**Operation:**

```
vAddr ← GPR[base] + GPR[index]
if vAddr₁..₀ ≠ 0² then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..3 || (pAddr₂..₀ xor (ReverseEndian || 0²))
memdoubleword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
bytesel ← vAddr₂..₀ xor (BigEndianCPU || 0²)
StoreFPR(ft, UNINTERPRETED_DOUBLEWORD,
        sign_extend(memdoubleword₃₁₊₈*bytesel..8*bytesel))
```

**Exceptions:**

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

| 31          26 | 25          21 | 20          16 | 15          11 | 10          6 | 5          0 |
|----------------|----------------|----------------|----------------|---------------|--------------|
| SPECIAL2       | rs             | rt             | 0              | 0             | MADD         |
| 011100         |                |                | 0000           | 00000         | 000000       |
| 6              | 5              | 5              | 5              | 5             | 6            |

**Format:** MADD rs, rt                                                                    **MIPS32**

**Purpose:**

To multiply two words and add the result to Hi, Lo

**Description:** (LO,HI) ← (rs x rt) + (LO,HI)

The 32-bit word value in GPR *rs* is multiplied by the 32-bit word value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is added to the 64-bit concatenated values of $HI_{31..0}$ and $LO_{31..0}$. The most significant 32 bits of the result are sign-extended and written into *HI* and the least signficant 32 bits are sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

**Restrictions:**

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← (HI_{31..0} || LO_{31..0}) + (GPR[rs]_{31..0} * GPR[rt]_{31..0})
HI ← sign_extend(temp_{63..32})
LO ← sign_extend(temp_{31..0})
```

**Exceptions:**

None

**Programming Notes:**

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

## Floating Point Multiply Add

**MADD.fmt**

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 3 | 2 0 |
|---|---|---|---|---|---|---|
| COP1X<br>010011 | fr | ft | fs | fd | MADD<br>100 | fmt |
| 6 | 5 | 5 | 5 | 5 | 3 | 3 |

**Format:**  MADD.S fd, fr, fs, ft          **MIPS64 (MIPS IV)**
MADD.D fd, fr, fs, ft          **MIPS64 (MIPS IV)**
MADD.PS fd, fr, fs, ft          **MIPS64 (MIPS V)**

**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 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*, and placed into FPR *fd*. The operands and result are values in format *fmt*.

MADD.PS multiplies then adds the upper and lower halves of FPR *fr*, FPR *fs*, and FPR *ft* independently, and ORs together any generated exceptional conditions.

*Cause* bits are ORed into the *Flag* bits if no exception is taken.

**Restrictions:**

The fields *fr*, *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPRE-DICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of MADD.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, vfr +_fmt (vfs ×_fmt vft))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

| | | | | | |
|---|---|---|---|---|---|
| **Multiply and Add Unsigned Word to Hi,Lo** | | | | | **MADDU** |

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| SPECIAL2 <br> 011100 | rs | rt | 0 <br> 00000 | 0 <br> 00000 | MADDU <br> 000001 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** MADDU rs, rt                                          **MIPS32**

**Purpose:**

To multiply two unsigned words and add the result to Hi, Lo.

**Description:** (LO,HI) ← (rs x rt) + (LO,HI)

The 32-bit word value in GPR *rs* is multiplied by the 32-bit word value in GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is added to the 64-bit concatenated values of $HI_{31..0}$ and $LO_{31..0}$. The most significant 32 bits of the result are sign-extended and written into *HI* and the least signficant 32 bits are sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

**Restrictions:**

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← (HI_31..0 || LO_31..0) + ((0^32 || GPR[rs]_31..0) * (0^32 || GPR[rt]_31..0))
HI ← sign_extend(temp_63..32)
LO ← sign_extend(temp_31..0)
```

**Exceptions:**

None

**Programming Notes:**

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

| 31 26 | 25 21 | 20 16 | 15 11 | 10 3 | 2 0 |
|---|---|---|---|---|---|
| COP0 010000 | MF 00000 | rt | rd | 0 00000000 | sel |
| 6 | 5 | 5 | 5 | 8 | 3 |

**Format:** MFC0 rt, rd **MIPS32**

**Purpose:**

To move the contents of a coprocessor 0 register to a general register.

**Description:** rt ← CPR[0,rd,sel]

The contents of the coprocessor 0 register specified by the combination of rd and sel are sign-extended and loaded into general register rt. Note that not all coprocessor 0 registers support the sel field. In those instances, the sel field must be zero.

**Restrictions:**

The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *rd* and *sel*.

**Operation:**

```
data ← CPR[0,rd,sel]₃₁..₀
GPR[rt] ← sign_extend(data)
```

data $\leftarrow$ CPR[0,rd,sel]$_{31..0}$
GPR[rt] $\leftarrow$ sign_extend(data)

**Exceptions:**

Coprocessor Unusable

Reserved Instruction

## Move Word From Floating Point

**MFC1**

| COP1 | MF | | | 0 |
|------|-----|------|------|------|
| 010001 | 00000 | rt | fs | 000 0000 0000 |
| 6 | 5 | 5 | 5 | 11 |

31          26  25          21  20          16  15          11  10          0
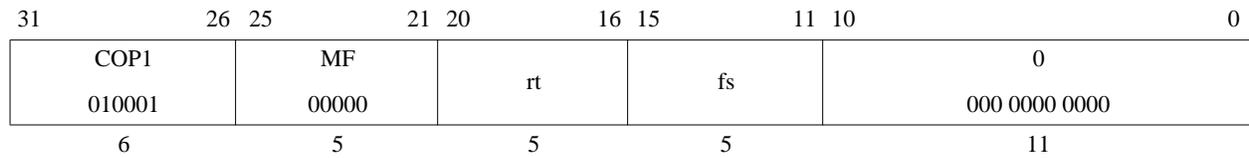
**Format:** MFC1 rt, fs

**MIPS32 (MIPS I)**

**Purpose:**

To copy a word from an FPU (CP1) general register to a GPR

**Description:** rt ← fs

The contents of FPR fs are sign-extended and loaded into general register rt.

**Restrictions:**

**Operation:**
```
data ← ValueFPR(fs, UNINTERPRETED_WORD)31..0
GPR[rt] ← sign_extend(data)
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Historical Information:**

For MIPS I, MIPS II, and MIPS III the contents of GPR *rt* are undefined for the instruction immediately following MFC1.

| 31        | 26 | 25    | 21 | 20 | 16 | 15 | 11 | 10          | 3 | 2   | 0 |
|-----------|----|-------|----|----|----|----|----|-------------|---|-----|---|
| COP2      |    | MF    |    | rt |    | rd |    | 0           |   | sel |   |
| 010010    |    | 00000 |    |    |    |    |    | 000 0000 0  |   |     |   |
| 6         |    | 5     |    | 5  |    | 5  |    | 8           |   | 3   |   |

**Format:**  MFC2 rt, rd                                                            **MIPS32**
             MFC2, rt, rd, sel                                                       **MIPS32**

**Purpose:**

To copy a word from a COP2 general register to a GPR

**Description:** rt ← rd

The contents of the lower 32-bits of GPR *rt* are sign-extended and placed into the coprocessor 2 register specified by the *rd* and *sel* fields. Note that not all coprocessor 2 registers may support the *sel* field. In those instances, the *sel* field must be zero.

**Restrictions:**

The results are **UNPREDICTABLE** is coprocessor 2 does not contain a register as specified by *rd* and *sel*.

**Operation:**

```
data ← CPR[2,rd,sel]₃₁..₀
GPR[rt] ← sign_extend(data)
```

**Exceptions:**

Coprocessor Unusable

| 31        26 | 25                16 | 15      11 | 10       6 | 5         0 |
|--------------|----------------------|------------|------------|-------------|
| SPECIAL | 0 | rd | 0 | MFHI |
| 000000 | 00 0000 0000 | | 00000 | 010000 |
| 6 | 10 | 5 | 5 | 6 |
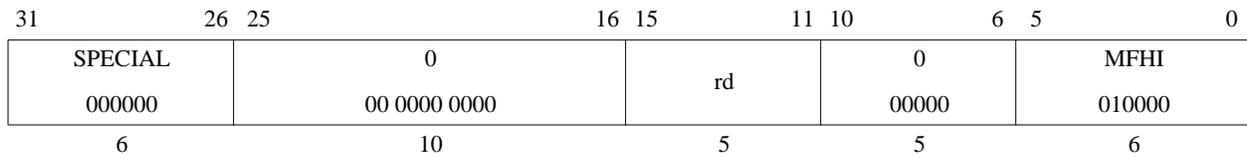
**Format:**  MFHI rd                                                                                    **MIPS32 (MIPS I)**

**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:**

None

**Operation:**

        GPR[rd] ← HI

**Exceptions:**

None

**Historical Information:**

In the MIPS I, II, and III architectures, the two instructions which follow the MFHI must not moodify the HI register. If this restriction is violated, the result of the MFHI is **UNPREDICTABLE**. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.

| 31 | 26 | 25 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|
| SPECIAL<br>000000 | | 0<br>00 0000 0000 | | rd | | 0<br>00000 | | MFLO<br>010010 | |
| 6 | | 10 | | 5 | | 5 | | 6 | |
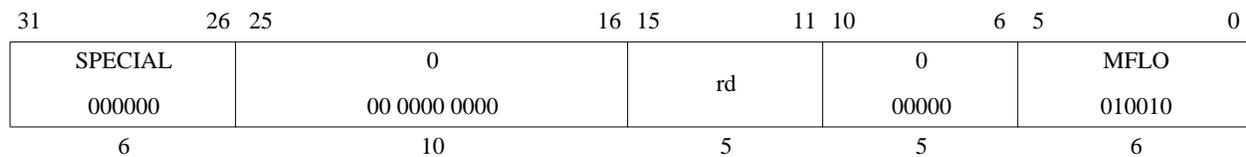
**Format:**   MFLO   rd                                                      **MIPS32 (MIPS I)**

**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: None**

**Operation:**

    GPR[rd] ← LO

**Exceptions:**

None

**Historical Information:**

In the MIPS I, II, and III architectures, the two instructions which follow the MFHI must not moodify the HI register. If this restriction is violated, the result of the MFHI is **UNPREDICTABLE**. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| COP1<br>010001 | | fmt | | 0<br>00000 | | fs | | fd | | MOV<br>000110 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |
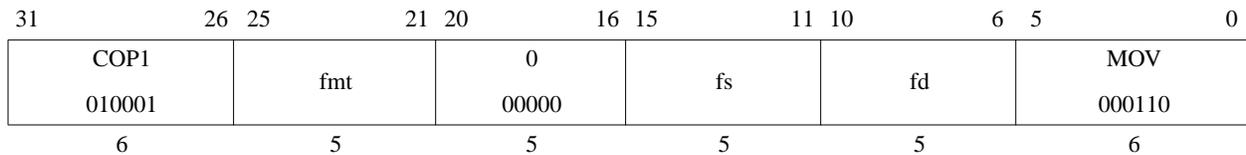
**Format:**  MOV.S fd, fs                                                                    **MIPS32 (MIPS I)**
MOV.D fd, fs                                                                    **MIPS32 (MIPS I)**
MOV.PS fd, fs                                                                   **MIPS64 (MIPS V)**

**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*. In paired-single format, both the halves of the pair are copied to *fd*.

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*; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of MOV.PS is undefined if the processor is executing in 16 FP registers mode.

**Operation:**

```
StoreFPR(fd, fmt, ValueFPR(fs, fmt))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation

| 31 | 26 | 25 | 21 | 20 | 18 | 17 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SPECIAL 000000 | | rs | | cc | | 0 0 | tf 0 | rd | | 0 00000 | | MOVCI 000001 | |
| 6 | | 5 | | 3 | | 1 | 1 | 5 | | 5 | | 6 | |

**Format:**  MOVF rd, rs, cc                                                                                    **MIPS32 (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:**

**Operation:**

```
if FPConditionCode(cc) = 0 then
    GPR[rd] ← GPR[rs]
endif
```

**Exceptions:**

Reserved Instruction, Coprocessor Unusable

**Floating Point Move Conditional on Floating Point False**                                    **MOVF.fmt**

| 31          26 | 25          21 | 20      18 | 17 | 16 | 15          11 | 10           6 | 5           0 |
|----------------|----------------|------------|----|----|----------------|----------------|---------------|
| COP1 010001 | fmt | cc | 0 0 | tf 0 | fs | fd | MOVCF 010001 |
| 6 | 5 | 3 | 1 | 1 | 5 | 5 | 6 |

**Format:**  MOVF.S  fd, fs, cc                                                    **MIPS32 (MIPS IV)**
MOVF.D  fd, fs, cc                                                    **MIPS32 (MIPS IV)**
MOVF.PS  fd, fs, cc                                                   **MIPS64 (MIPS V)**

**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* retains 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 **UNPREDICTABLE**.

MOVF.PS conditionally merges the lower half of FPR *fs* into the lower half of FPR *fd* if condition code *CC* is zero, and independently merges the upper half of FPR *fs* into the upper half of FPR *fd* if condition code *CC*+1 is zero. The *CC* field must be even; if it is odd, the result of this operation is **UNPREDICTABLE**.

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*; if they are not valid, the result is **UNPRE-DICTABLE**. The operand must be a value in format *fmt*; if it is not, the result is **UNPREDITABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of MOVF.PS is undefined if the processor is executing in 16 FP registers mode.

**Operation:**

```
if fmt ≠ PS
    if FPConditionCode(cc) = 0 then
        StoreFPR(fd, fmt, ValueFPR(fs, fmt))
    else
        StoreFPR(fd, fmt, ValueFPR(fd, fmt))
    endif
else
    mask ← 0
    if FPConditionCode(cc+0) = 0 then mask ← mask or 0xF0 endif
    if FPConditionCode(cc+1) = 0 then mask ← mask or 0x0F endif
    StoreFPR(fd, PS, ByteMerge(mask, fd, fs))
endif
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions***:*

Unimplemented Operation

**Move Conditional on Not Zero** <span style="float:right">**MOVN**</span>

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| SPECIAL<br>000000 | rs | rt | rd | 0<br>00000 | MOVN<br>001011 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** MOVN rd, rs, rt <span style="float:right">**MIPS32 (MIPS IV)**</span>

**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:**

None

**Programming Notes:**

The non-zero value tested here is the *condition true* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| COP1 010001 | | fmt | | rt | | fs | | fd | | MOVN 010011 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |
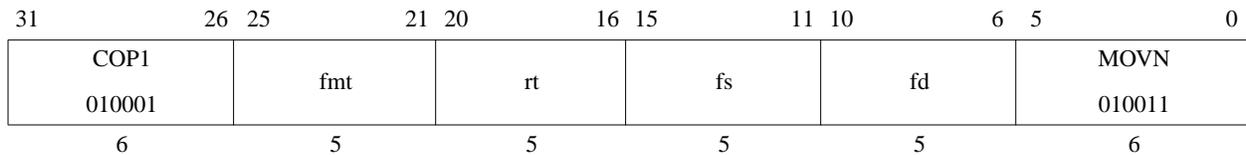
**Format:** MOVN.S fd, fs, rt **MIPS32 (MIPS IV)**
MOVN.D fd, fs, rt **MIPS32 (MIPS IV)**
MOVN.PS fd, fs, rt **MIPS64 (MIPS V)**

**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 **UNPREDICTABLE**.

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*; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of MOVN.PS is undefined if the processor is executing in 16 FP registers mode.

**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 Exceptions***:*

Unimplemented Operation

| 31 | | 26 | 25 | | 21 | 20 | | 18 | 17 | 16 | 15 | | 11 | 10 | | 6 | 5 | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SPECIAL | | | | rs | | | cc | | 0 | tf | | rd | | | 0 | | | MOVCI | |
| 000000 | | | | | | | | | 0 | 1 | | | | | 00000 | | | 000001 | |
| 6 | | | | 5 | | | 3 | | 1 | 1 | | 5 | | | 5 | | | 6 | |

**Format:**   MOVT rd, rs, cc                                          **MIPS32 (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:**

**Operation:**

```
if FPConditionCode(cc) = 1 then
    GPR[rd] ← GPR[rs]
endif
```

**Exceptions:**

Reserved Instruction, Coprocessor Unusable

**Floating Point Move Conditional on Floating Point True**                    **MOVT.fmt**

| 31 | 26 | 25 | 21 | 20 | 18 | 17 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| COP1 010001 | | fmt | | cc | | 0 0 | tf 1 | fs | | fd | | MOVCF 010001 | |
| 6 | | 5 | | 3 | | 1 | 1 | 5 | | 5 | | 6 | |

**Format:**   MOVT.S   fd, fs, cc                                    **MIPS32 (MIPS IV)**
             MOVT.D   fd, fs, cc                                    **MIPS32 (MIPS IV)**
             MOVT.PS  fd, fs, cc                                    **MIPS64 (MIPS V)**

**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 *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.

MOVT.PS conditionally merges the lower half of FPR *fs* into the lower half of FPR *fd* if condition code *CC* is one, and independently merges the upper half of FPR *fs* into the upper half of FPR *fd* if condition code *CC*+1 is one. The *CC* field should be even; if it is odd, the result of this operation is **UNPREDICTABLE**.

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*; if they are not valid, the result is **UNPRE-DICTABLE**. The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of MOVT.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
if fmt ≠ PS
    if FPConditionCode(cc) = 0 then
        StoreFPR(fd, fmt, ValueFPR(fs, fmt))
    else
        StoreFPR(fd, fmt, ValueFPR(fd, fmt))
    endif
else
    mask ← 0
    if FPConditionCode(cc+0) = 0 then mask ← mask or 0xF0 endif
    if FPConditionCode(cc+1) = 0 then mask ← mask or 0x0F endif
    StoreFPR(fd, PS, ByteMerge(mask, fd, fs))
endif
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation

**Move Conditional on Zero**                                                                        **MOVZ**

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

**Format:** MOVZ rd, rs, rt                                                            **MIPS32 (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:**
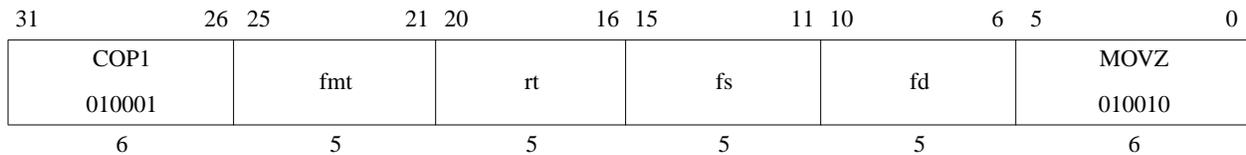
None

**Operation:**

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

**Exceptions:**

None

**Programming Notes:**

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

| 31          26 | 25          21 | 20       16 | 15       11 | 10        6 | 5              0 |
|----------------|----------------|-------------|-------------|-------------|------------------|
| COP1<br>010001 | fmt            | rt          | fs          | fd          | MOVZ<br>010010   |
| 6              | 5              | 5           | 5           | 5           | 6                |

**Format:**  MOVZ.S fd, fs, rt                                              **MIPS32 (MIPS IV)**
           MOVZ.D fd, fs, rt                                              **MIPS32 (MIPS IV)**
           MOVZ.PS fd, fs, rt                                              **MIPS64 (MIPS V)**

**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 *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 **UNPREDICTABLE**.

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*; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of MOVZ.PS is undefined if the processor is executing in 16 FP registers mode.

**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 Exceptions:*

Unimplemented Operation

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| SPECIAL2<br>011100 | rs | rt | 0<br>00000 | 0<br>00000 | MSUB<br>000100 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** MSUB rs, rt **MIPS32**

**Purpose:**

**To** multiply two words and subtract the result from Hi, Lo

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

The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values of $HI_{31..0}$ and $LO_{31..0}$. The most significant 32 bits of the result are sign-extended and written into *HI* and the least signficant 32 bits are sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

**Restrictions:**

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← (HI₃₁..₀ || LO₃₁..₀) - (GPR[rs]₃₁..₀ * GPR[rt]₃₁..₀)
HI ← sign_extend(temp₆₃..₃₂)
LO ← sign_extend(temp₃₁..₀)
```

**Exceptions:**

None

**Programming Notes:**

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

## Floating Point Multiply Subtract

<div align="right">

**MSUB.fmt**

</div>

| 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                                 **MIPS64 (MIPS IV)**
                 MSUB.D fd, fr, fs, ft                                 **MIPS64 (MIPS IV)**
                 MSUB.PS fd, fr, fs, ft                              **MIPS64 (MIPS V)**

### Purpose:

To perform a combined multiply-then-subtract 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 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*.

MSUB.PS multiplies then subtracts the upper and lower halves of FPR *fr*, FPR *fs*, and FPR *ft* independently, and ORs together any generated exceptional conditions.

*Cause* bits are ORed into the *Flag* bits if no exception is taken.

### Restrictions:

The fields *fr, fs, ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPRE-DICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of MSUB.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

### Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, (vfs ×fmt vft) −fmt vfr))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

| 31            | 26 | 25 | 21 | 20 | 16 | 15         | 11 | 10         | 6 | 5               | 0 |
|---------------|----|----|----|----|----|------------|----|------------|---|-----------------|---|
| SPECIAL2      |    | rs |    | rt |    | 0          |    | 0          |   | MSUBU           |   |
| 011100        |    |    |    |    |    | 00000      |    | 00000      |   | 000101          |   |
| 6             |    | 5  |    | 5  |    | 5          |    | 5          |   | 6               |   |

**Format:** MSUBU rs, rt **MIPS32**

**Purpose:**

To multiply two words and subtract the result from Hi, Lo

**Description:** (LO,HI) ← (rs x rt) - (LO,HI)

The 32-bit word value in GPR *rs* is multiplied by the 32-bit word value in GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values of $HI_{31..0}$ and $LO_{31..0}$. The most significant 32 bits of the result are sign-extended and written into *HI* and the least signficant 32 bits are sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

**Restrictions:**

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← (HI31..0 || LO31..0) - ((0^32 || GPR[rs]31..0) * (0^32 || GPR[rt]31..0))
HI ← sign_extend(temp63..32)
LO ← sign_extend(temp31..0)
```

**Exceptions:**

None

**Programming Notes:**

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 3 | 2 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COP0 010000 | | MT 00100 | | rt | | rd | | 0 0000 000 | | sel | |
| 6 | | 5 | | 5 | | 5 | | 8 | | 3 | |

**Format:** MTC0 rt, rd **MIPS32**

**Purpose:**

To move the contents of a general register to a coprocessor 0 register.

**Description:** CPR[r0, rd, sel] ← rt

The contents of general register rt are loaded into the coprocessor 0 register specified by the combination of rd and sel. Not all coprocessor 0 registers support the the sel field. In those instances, the sel field must be set to zero.

**Restrictions:**

The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *rd* and *sel*.
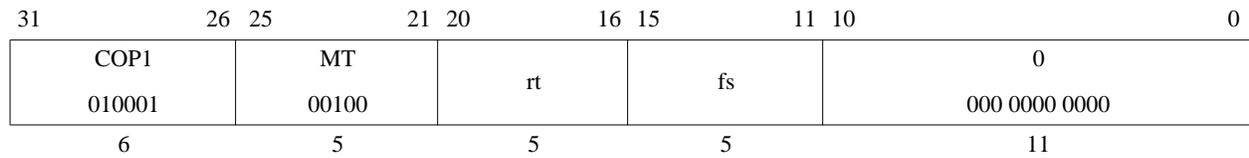
**Operation:**

```
if (Width(CPR[0,rd,sel]) = 64) then
    CPR[0,rd,sel] ← data
else
    CPR[0,rd,sel] ← data31..0
endif
```

**Exceptions:**

Coprocessor Unusable

Reserved Instruction

| 31          26 | 25          21 | 20        16 | 15        11 | 10                                    0 |
|:--------------:|:--------------:|:------------:|:------------:|:---------------------------------------:|
| COP1<br>010001 | MT<br>00100 | rt | fs | 0<br>000 0000 0000 |
| 6 | 5 | 5 | 5 | 11 |

**Format:**  MTC1 rt, fs                                                                               **MIPS32 (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.

**Restrictions:**

**Operation:**

```
data ← GPR[rt]₃₁..₀
StoreFPR(fs, UNINTERPRETED_WORD, data)
```

**Exceptions:**

Coprocessor Unusable

**Historical Information:**

For MIPS I, MIPS II, and MIPS III the value of FPR *fs* is UNPREDICTABLE for the instruction immediately following MTC1.

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|
| COP2 010010 | | MT 00100 | | rt | | rd | | 0 000 0000 0 | | sel |
| 6 | | 5 | | 5 | | 5 | | 8 | | 3 |

**Format:**   MTC2 rt, rd                                                                                **MIPS32**
           MTC2 rt, rd, sel                                                             **MIPS32**

**Purpose:**

To copy a word from a GPR to a COP2 general register

**Description:** rd ← rt

The low word in GPR *rt* is placed into the low word of coprocessor 2 general register specified by the *rd* and *sel* fields. If coprocessor 2 general registers are 64 bits wide, bits 63..32 of register *rd* become undefined. Note that not all coprocessor 2 registers may support the *sel* field. In those instances, the *sel* field must be zero.
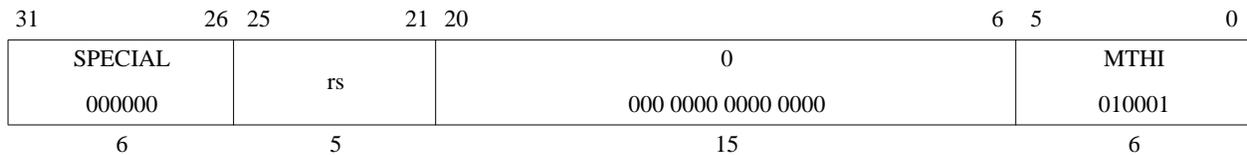
**Restrictions:**

The results are **UNPREDICTABLE** is coprocessor 2 does not contain a register as specified by *rd* and *sel*.

**Operation:**

```
data ← GPR[rt]₃₁..₀
CPR[2,rd,sel] ← data
```

**Exceptions:**

Coprocessor Unusable

## Move to HI Register

| 31 26 | 25 21 | 20 6 | 5 0 |
|---|---|---|---|
| SPECIAL<br>000000 | rs | 0<br>000 0000 0000 0000 | MTHI<br>010001 |
| 6 | 5 | 15 | 6 |

**Format:** MTHI rs                                                                                      **MIPS32 (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:**

A computed result written to the *HI/LO* pair by DIV, DIVU, DDIV, DDIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either *HI* or *LO*.

If an MTHI instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of *LO* are UNPREDICTABLE. 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 UNPREDICTABLE value
```
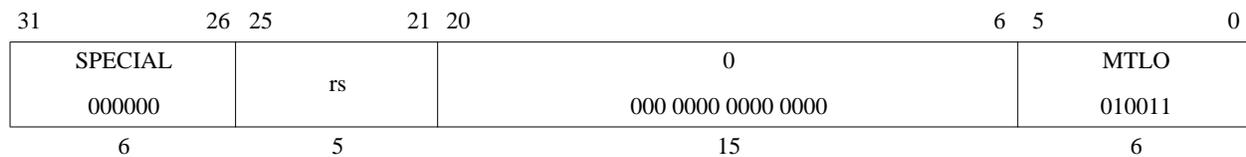
**Operation:**

```
HI ← GPR[rs]
```

**Exceptions:**

None

**Historical Information:**

In MIPS I-III, if either of the two preceding instructions is MFHI, the result of that MFHI is UNPREDICTABLE. Reads of the *HI* or *LO* special register must be separated from any subsequent instructions that write to them by two or more instructions. In MIPS IV and later, including MIPS32 and MIPS64, this restriction does not exist.

| 31          26 | 25          21 | 20                              6 | 5              0 |
|----------------|----------------|-----------------------------------|------------------|
| SPECIAL        |                | 0                                 | MTLO             |
| 000000         | rs             | 000 0000 0000 0000                | 010011           |
| 6              | 5              | 15                                | 6                |

**Format:**  MTLO rs                                                                             **MIPS32 (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:**

A computed result written to the *HI/LO* pair by DIV, DIVU, DDIV, DDIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either *HI* or *LO*.

If an MTLO instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of *HI* are UNPREDICTABLE. 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 UNPREDICTABLE value
```

**Operation:**

```
LO ← GPR[rs]
```

**Exceptions:**

None

**Historical Information:**

In MIPS I-III, if either of the two preceding instructions is MFHI, the result of that MFHI is UNPREDICTABLE. Reads of the *HI* or *LO* special register must be separated from any subsequent instructions that write to them by two or more instructions. In MIPS IV and later, including MIPS32 and MIPS64, this restriction does not exist.

**Multiply Word to GPR**                                                                                      **MUL**

| 31          26 | 25        21 | 20      16 | 15      11 | 10        6 | 5          0 |
|----------------|--------------|------------|------------|-------------|--------------|
| SPECIAL2       | rs           | rt         | rd         | 0           | MUL          |
| 011100         |              |            |            | 00000       | 000010       |
| 6              | 5            | 5          | 5          | 5           | 6            |

**Format:**  MUL rd, rs, rt                                                                                   **MIPS32**

**Purpose:**

To multiply two words and write the result to a GPR.

**Description:** rd ← rs × rt

The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The least significant 32 bits of the product are sign-extended and written to GPR *rd*. The contents of *HI* and *LO* are **UNPREDICTABLE** after the operation. No arithmetic exception occurs under any circumstances.

**Restrictions:**

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

Note that this instruction does not provide the capability of writing the result to the HI and LO registers.

**Operation:**

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
    UndefinedResult()
endif
temp <- GPR[rs] * GPR[rt]
GPR[rd] <- sign_extend(temp_{31..0})
HI <- UNPREDICTABLE
LO <- UNPREDICTABLE
```
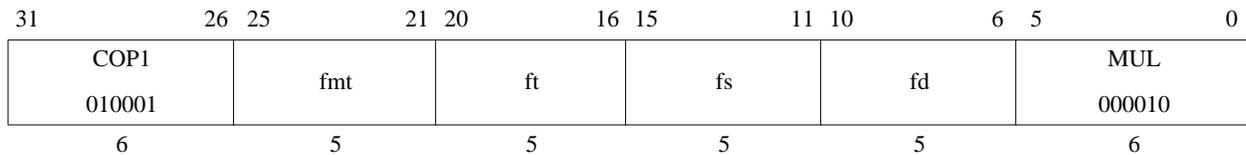
**Exceptions:**

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 interlocks 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.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| COP1 010001 | fmt | ft | fs | fd | MUL 000010 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**   MUL.S  fd, fs, ft     **MIPS32 (MIPS I)**
            MUL.D  fd, fs, ft     **MIPS32 (MIPS I)**
            MUL.PS fd, fs, ft     **MIPS64 (MIPS V)**

**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*. MUL.PS multiplies the upper and lower halves of FPR *fs* and FPR *ft* independently, and ORs together any generated exceptional conditions.

**Restrictions:**

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of MUL.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.
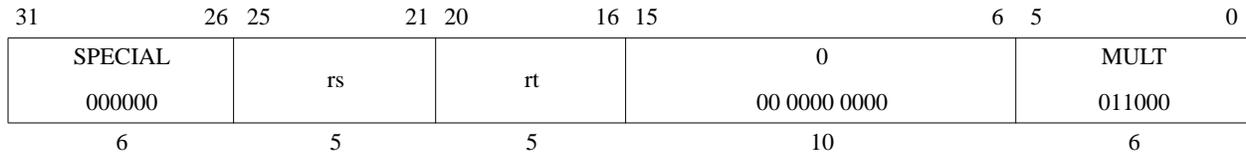
**Operation:**

    StoreFPR (fd, fmt, ValueFPR(fs, fmt) ×$_{fmt}$ ValueFPR(ft, fmt))

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

**Multiply Word**                                                                                                                                          **MULT**

| 31        26 | 25        21 | 20        16 | 15                          6 | 5              0 |
|:---:|:---:|:---:|:---:|:---:|
| SPECIAL<br>000000 | rs | rt | 0<br>00 0000 0000 | MULT<br>011000 |
| 6 | 5 | 5 | 10 | 6 |

**Format:** MULT rs, rt                                                                                                            **MIPS32 (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 sign-extended and placed into special register *LO*, and the high-order 32-bit word is sign-extended and placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

**Restrictions:**

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

**Operation:**

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
   UndefinedResult()
endif
   prod   ← GPR[rs]₃₁..₀ × GPR[rt]₃₁..₀
   LO     ← sign_extend(prod₃₁..₀)
   HI     ← sign_extend(prod₆₃..₃₂)
```
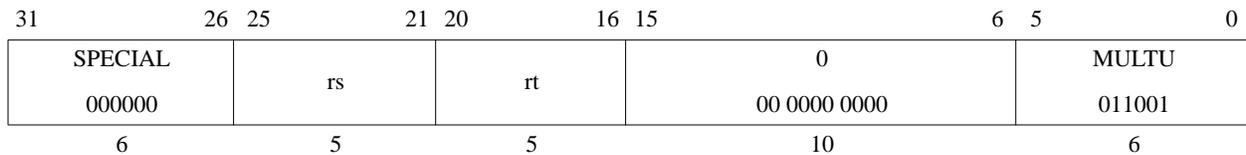
**Exceptions:**

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 interlocks 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.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| SPECIAL<br>000000 | | rs | | rt | | 0<br>00 0000 0000 | | MULTU<br>011001 | |
| 6 | | 5 | | 5 | | 10 | | 6 | |

**Format:** MULTU rs, rt **MIPS32 (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 sign-extended and placed into special register *LO*, and the high-order 32-bit word is sign-extended and placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

**Restrictions:**

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

**Operation:**

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UndefinedResult()
endif
   prod← (0 || GPR[rs]31..0) × (0 || GPR[rt]31..0)
   LO ← sign_extend(prod31..0)
   HI ← sign_extend(prod63..32)
```
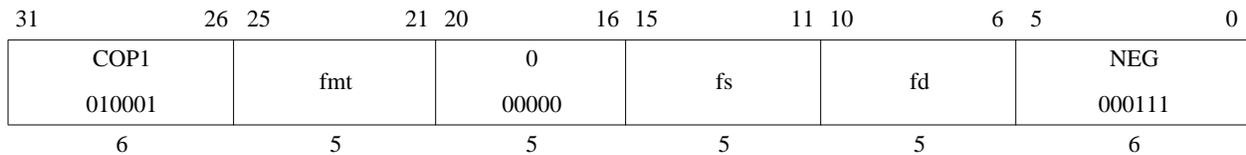
**Exceptions:**

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 interlocks 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.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

## Floating Point Negate — NEG.fmt

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| COP1 010001 | | fmt | | 0 00000 | | fs | | fd | | NEG 000111 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:**
| | |
|---|---|
| NEG.S fd, fs | **MIPS32 (MIPS I)** |
| NEG.D fd, fs | **MIPS32 (MIPS I)** |
| NEG.PS fd, fs | **MIPS64 (MIPS V)** |

### 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*. NEG.PS negates the upper and lower halves of FPR *fs* independently, and ORs together any generated exceptional conditions.

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*; if they are not valid, the result is **UNPREDICTABLE**. The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of NEG.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

### Operation:

```
StoreFPR(fd, fmt, Negate(ValueFPR(fs, fmt)))
```

### Exceptions:

Coprocessor Unusable, Reserved Instruction

### Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 3 | 2 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| COP1X 010011 | | fr | | ft | | fs | | fd | | NMADD 110 | | fmt | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 3 | | 3 | |

**Format:**  NMADD.S  fd, fr, fs, ft                                **MIPS64 (MIPS IV)**
            NMADD.D  fd, fr, fs, ft                                **MIPS64 (MIPS IV)**
            NMADD.PS  fd, fr, fs, ft                              **MIPS64 (MIPS V)**

**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*.

NMADD.PS applies the operation to the upper and lower halves of FPR *fr*, FPR *fs*, and FPR *ft* independently, and ORs together any generated exceptional conditions.

*Cause* bits are ORed into the *Flag* bits if no exception is taken.

**Restrictions:**

The fields *fr*, *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPRE-DICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of NMADD.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, −(vfr +_fmt (vfs ×_fmt vft)))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

| 31          26 | 25        21 | 20      16 | 15      11 | 10       6 | 5    3 | 2   0 |
|----------------|--------------|------------|------------|------------|--------|-------|
| COP1X<br>010011 | fr | ft | fs | fd | NMSUB<br>111 | fmt |
| 6 | 5 | 5 | 5 | 5 | 3 | 3 |

**Format:**  NMSUB.S fd, fr, fs, ft                                            **MIPS64 (MIPS IV)**
            NMSUB.D fd, fr, fs, ft                                            **MIPS64 (MIPS IV)**
            NMSUB.PS fd, fr, fs, ft                                           **MIPS64 (MIPS V)**

**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*.

NMSUB.PS applies the operation to the upper and lower halves of FPR *fr*, FPR *fs*, and FPR *ft* independently, and ORs together any generated exceptional conditions.

*Cause* bits are ORed into the *Flag* bits if no exception is taken.

**Restrictions:**

The fields *fr*, *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of NMSUB.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, −((vfs ×_fmt vft) −_fmt vfr))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

| 31        26 | 25        21 | 20        16 | 15        11 | 10        6 | 5        0 |
|:---:|:---:|:---:|:---:|:---:|:---:|
| SPECIAL | 0 | 0 | 0 | 0 | SLL |
| 000000 | 00000 | 00000 | 00000 | 00000 | 000000 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** NOP                                                                     **Assembly Idiom**

**Purpose:**

To perform no operation.

**Description:**

NOP is the assembly idiom used to denote no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 0.

**Restrictions:**

None

**Operation:**

None

**Exceptions:**

None

**Programming Notes:**

The zero instruction word, which represents SLL, r0, r0, 0, is the preferred NOP for software to use to fill branch and jump delay slots and to pad out alignment sequences.

| 31           26 | 25        21 | 20        16 | 15        11 | 10         6 | 5            0 |
|-----------------|--------------|--------------|--------------|--------------|----------------|
| SPECIAL<br>000000 | rs | rt | rd | 0<br>00000 | NOR<br>100111 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** NOR rd, rs, rt                                                               **MIPS32 (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 000000 | | rs | | rt | | rd | | 0 00000 | | OR 100101 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |
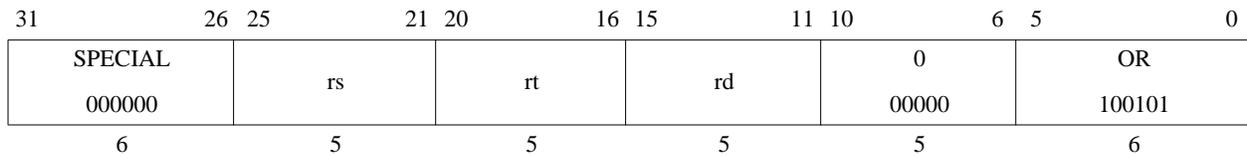
**Format:** OR rd, rs, rt **MIPS32 (MIPS I)**

**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** <span style="float:right">**ORI**</span>

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|----|----|----|----|----|----|----|----|
| ORI<br>001101 | | rs | | rt | | immediate | |
| 6 | | 5 | | 5 | | 16 | |
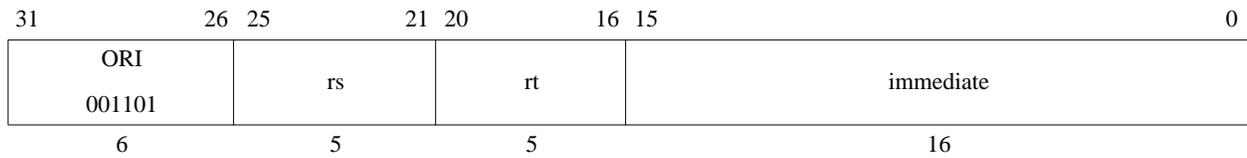
**Format:** `ORI rt, rs, immediate`  <span style="float:right">**MIPS32 (MIPS I)**</span>

**Purpose:**

To do a bitwise logical OR with a constant

**Description:** `rt ← 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] ← GPR[rs] or zero_extend(immediate)
```

**Exceptions:**

None

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| COP1 010001 | | fmt 10110 | | ft | | fs | | fd | | PLL 101100 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:** PLL.PS fd, fs, ft                                     **MIPS64 (MIPS V)**

**Purpose:**

To merge a pair of paired single values with realignment

**Description:** fd ← lower(fs) || lower(ft)

A new paired-single value is formed by catenating the lower single of *fs* (bits 31..0) and the lower single of *ft* (bits 31..0).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields *fs, ft,* and *fd* must specify FPRs valid for operands of type *PS*. If they are not valid, the result is **UNPRE-DICTABLE**.
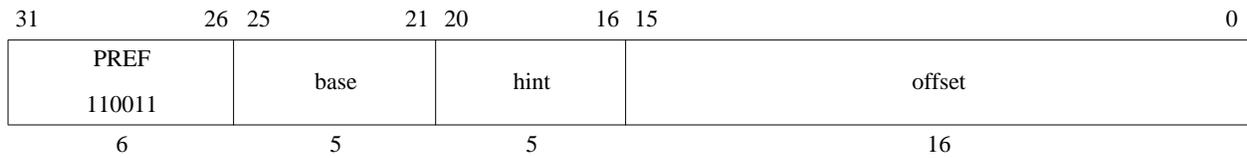
The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

StoreFPR(fd, PS, ValueFPR(fs, PS)$_{31..0}$ || ValueFPR(ft, PS)$_{31..0}$)

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Pair Lower Upper**　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　**PLU.PS**

| 31      26 | 25      21 | 20      16 | 15      11 | 10       6 | 5        0 |
|------------|------------|------------|------------|------------|------------|
| COP1 | fmt | ft | fs | fd | PLU |
| 010001 | 10110 | | | | 101101 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**　`PLU.PS fd, fs, ft`　　　　　　　　　　　　　　　　　　　**MIPS64 (MIPS V)**

**Purpose:**

To merge a pair of paired single values with realignment

**Description:** `fd ← lower(fs) || upper(ft)`

A new paired-single value is formed by catenating the lower single of *fs* (bits 31..0) and the upper single of *ft* (bits 63..32).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields *fs, ft,* and *fd* must specify FPRs valid for operands of type *PS*. If they are not valid, the result is **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

$$\text{StoreFPR(fd, PS, ValueFPR(fs, PS)}_{31..0} \text{ || ValueFPR(ft, PS)}_{63..32})$$

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|
| PREF<br>110011 | | base | | hint | | offset | |
| 6 | | 5 | | 5 | | 16 | |

**Format:** `PREF hint,offset(base)` **MIPS32 (MIPS IV)**

**Purpose:**

To move data between memory and cache.

**Description:** `prefetch_memory(base+offset)`

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

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 as long as it does not change architecturally visible state or alter the meaning of a program. Implementations are expected either to do nothing, or to take an action that increases the performance of the program.

PREF does not cause addressing-related exceptions. If the address specified would cause an addressing exception, the exception condition is ignored and no data movement occurs.However even if no data is prefetched, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

PREF never generates a memory operation for a location with an *uncached* memory access type.

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.

For a cached location, the expected and useful action for the processor is 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. A *hint* value cannot cause an action to modify architecturally visible state. A processor may use a *hint* value to improve the effectiveness of the prefetch action.

**Table 3-29 Values of the *hint* Field for the PREF Instruction**

| Value | Name | Data Use and Desired Prefetch Action |
|---|---|---|
| 0 | load | Use: Prefetched data is expected to be read (not modified).<br><br>Action: Fetch data as if for a load. |
| 1 | store | Use: Prefetched data is expected to be stored or modified.<br><br>Action: Fetch data as if for a store. |
| 2-3 | Reserved | Reserved for future use - not available to implementations. |
| 4 | load_streamed | Use: Prefetched data is expected to be read (not modified) but not reused extensively; it "streams" through cache.<br><br>Action: Fetch data as if for a load and place it in the cache so that it does not displace data prefetched as "retained." |
| 5 | store_streamed | Use: Prefetched data is expected to be stored or modified but not reused extensively; it "streams" through cache.<br><br>Action: Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as "retained." |
| 6 | load_retained | Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be "retained" in the cache.<br><br>Action: Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as "streamed." |
| 7 | store_retained | Use: Prefetched data is expected to be stored or modified and reused extensively; it should be "retained" in the cache.<br><br>Action: Fetch data as if for a store and place it in the cache so that it is not displaced by data prefetched as "streamed." |

**Table 3-29 Values of the *hint* Field for the PREF Instruction**

| 8-24 | Reserved | Reserved for future use - not available to implementations. |
|---|---|---|
| 25 | writeback_invalidate (also known as "nudge") | Use: Data is no longer expected to be used.<br><br>Action: For a writeback cache, schedule a wirteback of any dirty data. At the completion of the writeback, mark the state of any cache lines written back as invalid. |
| 26-29 | Implementation Dependent | Unassigned by the Architecture - available for implementation-dependent use. |
| 30 | PrepareForStore | Use: Prepare the cache for writing an entire line, without the overhead involved in filling the line from memory.<br><br>Action: If the reference hits in the cache, no action is taken. If the reference misses in the cache, a line is selected for replacement, any valid and dirty victim is written back to memory, the entire line is filled with zero data, and the state of the line is marked as valid and dirty. |
| 31 | Implementation Dependent | Unassigned by the Architecture - available for implementation-dependent use. |

**Restrictions:**

None

**Operation:**

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

**Exceptions:**

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

**Programming Notes:**

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

*Hint* field encodings whose function is described as "streamed" or "retained" convey usage intent from software to hardware. Software should not assume that hardware will always prefetch data in an optimal way. If data is to be truly retained, software should use the Cache instruction to lock data into the cache.

## Prefetch Indexed

<div align="right">

**PREFX**

</div>

| 31          | 26 25 | 21 20 | 16 15 | 11 10      | 6 5             | 0 |
|-------------|-------|-------|-------|------------|-----------------|---|
| COP1X 010011 | base  | index | hint  | 0 00000    | PREFX 001111    |   |
| 6           | 5     | 5     | 5     | 5          | 6               |   |
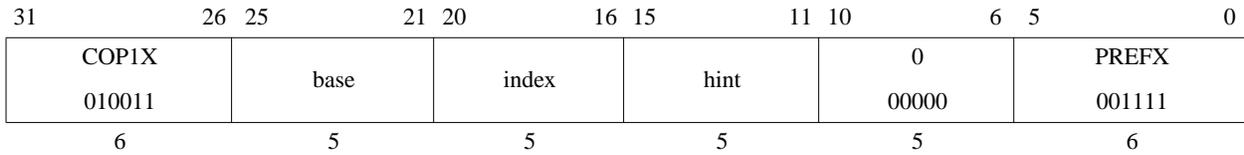
**Format:** `PREFX hint, index(base)`  **MIPS64 (MIPS IV)**

**Purpose:**

To move data between memory and cache.

**Description:** `prefetch_memory[base+index]`

PREFX adds the contents of GPR *index* to the contents of GPR *base* to form an effective byte address. The *hint* field supplies information about the way the data is expected to be used.

The only functional difference between the PREF and PREFX instructions is the addressing mode implemented by the two. Refer to the PREF instruction for all other details, including the encoding of the *hint* field.

**Restrictions:**

**Operation:**

```
vAddr ← GPR[base] + GPR[index]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Programming Notes:**

The PREFX instruction is only available on processors that implement floating point and should never by generated by compilers in situations in which the corresponding load and store indexed floating point instructions are generated.

Also refer to the corresponding section in the PREF instruction description.

**Pair Upper Lower**                                                                    **PUL.PS**

| 31      26 | 25      21 | 20      16 | 15      11 | 10      6 | 5      0 |
|------------|------------|------------|------------|-----------|----------|
| COP1       | fmt        | ft         | fs         | fd        | PUL      |
| 010001     | 10110      |            |            |           | 101110   |
| 6          | 5          | 5          | 5          | 5         | 6        |

**Format:** PUL.PS fd, fs, ft                                                    **MIPS64 (MIPS V)**

**Purpose:**

To merge a pair of paired single values with realignment

**Description:** fd ← upper(fs) || lower(ft)

A new paired-single value is formed by catenating the upper single of *fs* (bits 63..32) and the lower single of *ft* (bits 31..0).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields *fs, ft,* and *fd* must specify FPRs valid for operands of type *PS*. If they are not valid, the result is **UNPRE-DICTABLE**.
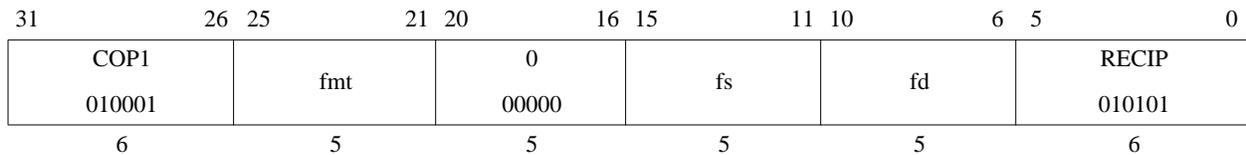
The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

StoreFPR(fd, PS, ValueFPR(fs, PS)$_{63..32}$ || ValueFPR(ft, PS)$_{31..0}$)

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| COP1<br>010001 | fmt<br>10110 | ft | fs | fd | PUU<br>101111 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** PUU.PS fd, fs, ft **MIPS64 (MIPS V)**

**Purpose:**

To merge a pair of paired single values with realignment

**Description:** fd ← upper(fs) || upper(ft)

A new paired-single value is formed by catenating the upper single of *fs* (bits 63..32) and the upper single of *ft* (bits 63..32).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields *fs, ft,* and *fd* must specify FPRs valid for operands of type *PS*. If they are not valid, the result is **UNPRE-DICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.
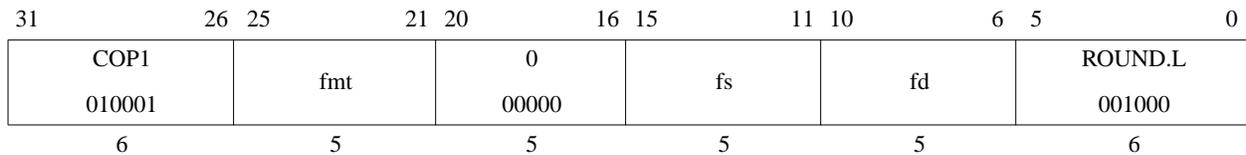
**Operation:**

StoreFPR(fd, PS, ValueFPR(fs, PS)$_{63..32}$ || ValueFPR(ft, PS)$_{63..32}$)

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| COP1 010001 | | fmt | | 0 00000 | | fs | | fd | | RECIP 010101 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:** RECIP.S   fd, fs                                    **MIPS64 (MIPS IV)**
RECIP.D   fd, fs                                    **MIPS64 (MIPS IV)**

**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*; if they are not valid, the result is **UNPRE-DICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of RECIP.D is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
StoreFPR(fd, fmt, 1.0 / valueFPR(fs, fmt))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Division-by-zero, Unimplemented Op, Invalid Op, Overflow, Underflow

| 31        26 | 25        21 | 20        16 | 15        11 | 10        6 | 5        0 |
|---|---|---|---|---|---|
| COP1<br>010001 | fmt | 0<br>00000 | fs | fd | ROUND.L<br>001000 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**  ROUND.L.S   fd, fs                                    **MIPS64 (MIPS III)**
          ROUND.L.D   fd, fs                                    **MIPS64 (MIPS III)**

**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 and rounded 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. In this case 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*.

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

**Exceptions:**

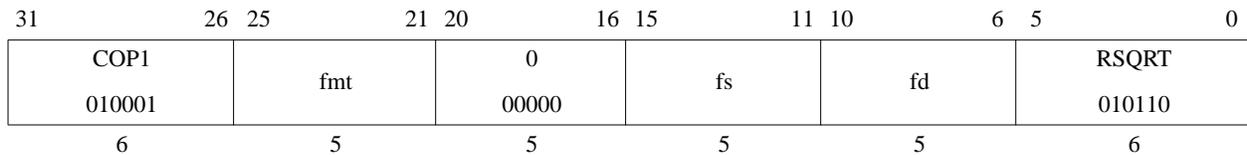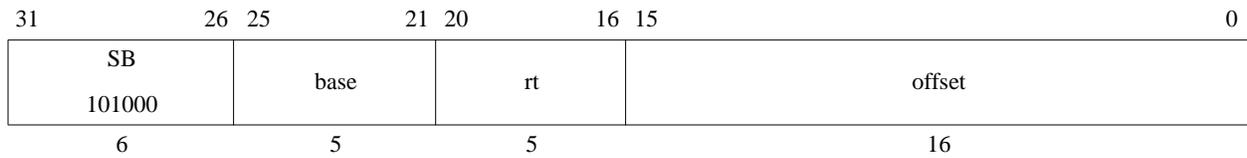Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Unimplemented Operation, Invalid Operation, Overflow

## Floating Point Round to Word Fixed Point

**ROUND.W.fmt**

| 31      26 | 25      21 | 20      16 | 15      11 | 10      6 | 5      0 |
|------------|------------|------------|------------|-----------|----------|
| COP1<br>010001 | fmt | 0<br>00000 | fs | fd | ROUND.W<br>001100 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**  ROUND.W.S   fd, fs                                **MIPS32 (MIPS II)**
             ROUND.W.D   fd, fs                                **MIPS32 (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. In this case 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*.

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

**Operation:**

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Unimplemented Operation, Invalid Operation, Overflow

| 31          26 | 25          21 | 20          16 | 15          11 | 10          6 | 5          0 |
|----------------|----------------|----------------|----------------|---------------|--------------|
| COP1<br>010001 | fmt            | 0<br>00000     | fs             | fd            | RSQRT<br>010110 |
| 6              | 5              | 5              | 5              | 5             | 6            |

**Format:**   RSQRT.S   fd, fs                                    **MIPS64 (MIPS IV)**
            RSQRT.D   fd, fs                                    **MIPS64 (MIPS IV)**

**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 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).

The effect of the current *FCSR* rounding mode on the result is implementation dependent.

**Restrictions:**

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPRE-DICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of RSQRT.D is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

```
StoreFPR(fd, fmt, 1.0 / SquareRoot(valueFPR(fs, fmt)))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Division-by-zero, Unimplemented Operation, Invalid Operation, Overflow, Underflow

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|
| SB<br>101000 | | base | | rt | | offset | |
| 6 | | 5 | | 5 | | 16 | |

**Format:** SB rt, offset(base)   **MIPS32 (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:**
```
vAddr      ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr      ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor ReverseEndian^3)
bytesel    ← vAddr_2..0 xor BigEndianCPU^3
datadoubleword← GPR[rt]_63-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, BYTE, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error

| 31 | | 26 | 25 | | 21 | 20 | | 16 | 15 | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SC<br>111000 | | | base | | | rt | | | offset | | |
| 6 | | | 5 | | | 5 | | | 16 | | |
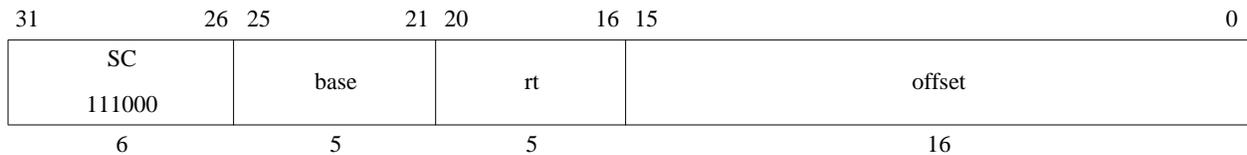
**Format:** SC rt, offset(base)                                    **MIPS32 (MIPS II)**

**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. To complete the RMW sequence atomically, the following occur:

- The least-significant 32-bit word of GPR *rt* is stored into memory at the location specified by the aligned effective address.

- A 1, indicating success, is written into GPR *rt*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *rt*.

If either of the following events occurs between the execution of LL and SC, the SC fails:

- 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, but it is at least one word and at most the minimum page size.

- An exception occurs on the processor executing the LL/SC.

If either 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 not predictable. 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 is 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:

- **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, and 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.

**Restrictions:**

The addressed location must have a memory access type of *cached noncoherent* or *cached coherent*; if it does not, the result is undefined.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr₁..₀ ≠ 0² then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr₂..₀ xor (ReverseEndian || 0²))
bytesel← vAddr₂..₀ xor (BigEndianCPU || 0²)
datadoubleword← GPR[rt]₆₃₋₈*bytesel..0 || 0^8*bytesel
if LLbit then
    StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
endif
GPR[rt]← 0⁶³ || 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 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, and 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.

## Store Conditional Doubleword SCD

| 31 26 | 25 21 | 20 16 | 15 0 |
|---|---|---|---|
| SCD<br>111100 | base | rt | offset |
| 6 | 5 | 5 | 16 |
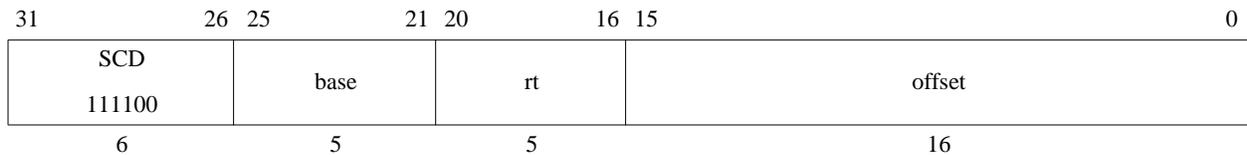
**Format:** SCD rt, offset(base) **MIPS64 (MIPS III)**

**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, the following occur**:**

• The 64-bit doubleword of GPR *rt* is stored into memory at the location specified by the aligned effective address.

• A 1, indicating success, is written into GPR *rt*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *rt*.

If either of the following events occurs between the execution of LLD and SCD, the SCD fails:

• Another processor completes a coherent store or a coherent I/O module into the block of physical memory containing the word. The size and alignment of the block is implementation dependent, but it is at least one doubleword and 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:

• detect exceptions and fail when an exception occurs

• fail after the return-from-interrupt instruction (RFE or ERET) is executed

• both of the above

If either of the following events occurs between the execution of LLD and SCD, the SCD may succeed or it may fail; success or failure is not predictable. Portable programs should not cause 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 two conditions must be true or the result of the SCD is undefined:

• Execution of the SCD must be preceded by execution of an LLD instruction.

• An 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:

- **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, and 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.

This section 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 can 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. The 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 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr₂..₀ ≠ 0³ then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
datadoubleword ← GPR[rt]
if LLbit then
    StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 0⁶³ || LLbit
```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Address Error, Reserved Instruction
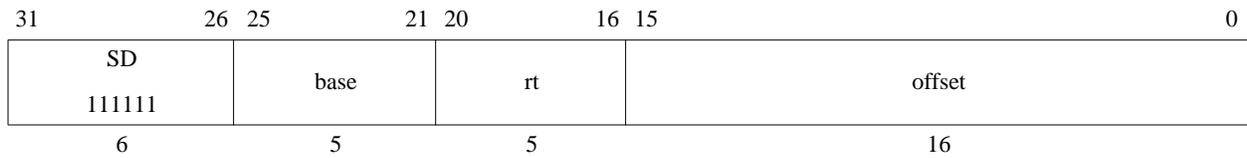
**Programming Notes:**

LLD and SCD are used to atomically update memory locations, as shown 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 such exceptions are arithmetic operations that trap, system calls, and 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.

| Store Doubleword | SD |
|---|---|

| 31      26 | 25      21 | 20      16 | 15      0 |
|---|---|---|---|
| SD<br>111111 | base | rt | offset |
| 6 | 5 | 5 | 16 |

**Format:** SD rt, offset(base)                                              **MIPS64 (MIPS III)**

**Purpose:**

To store a doubleword to memory

**Description:** memory[base+offset] ← 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 3 least-significant bits of the effective address is non-zero, an Address Error exception occurs.
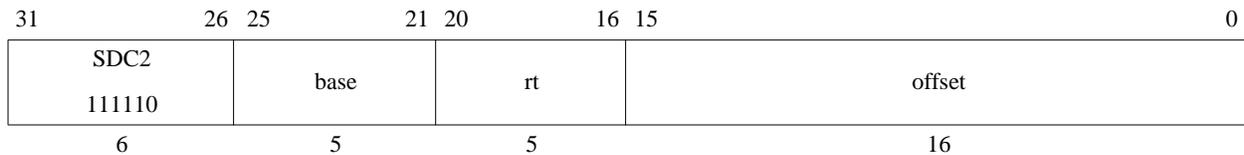
**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 0³ then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
datadoubleword← GPR[rt]
StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

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

| 31 | 26 | 25 | 6 | 5 | 0 |
|---|---|---|---|---|---|
| SPECIAL2<br>011100 | | code | | | SDBBP<br>111111 |
| 6 | | 20 | | | 6 |

**Format:** SDBBP code **EJTAG**

**Purpose:**

To cause a debug breakpoint exception

**Description:**

This instruction causes a debug exception, passing control to the debug exception handler. The code field can be used for passing information to the debug exception handler, and is retrieved by the debug exception handler only by loading the contents of the memory word containing the instruction, using the DEPC register. The CODE field is not used in any way by the hardware.

**Restrictions:**

**Operation:**

```
If Debug_DM = 0 then
    SignalDebugBreakpointException()
else
    SignalDebugModeBreakpointException()
endif
```

**Exceptions:**

Debug Breakpoint Exception

**Store Doubleword from Floating Point** <span style="float:right">**SDC1**</span>

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|----|----|----|----|----|----|----|---|
| SDC1<br>111101 | | base | | ft | | offset | |
| 6 | | 5 | | 5 | | 16 | |
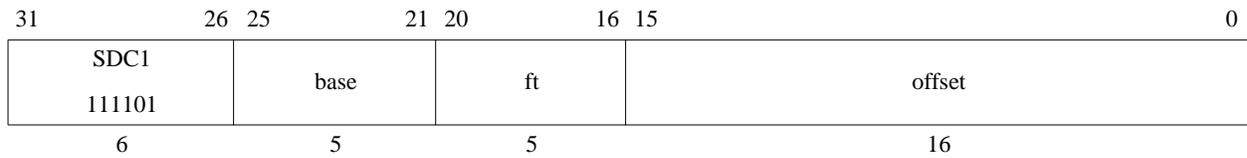
**Format:** SDC1 ft, offset(base) <span style="float:right">**MIPS32 (MIPS II)**</span>

**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.

**Restrictions:**

An Address Error exception occurs if EffectiveAddress$_{2..0}$ ≠ 0 (not doubleword-aligned).
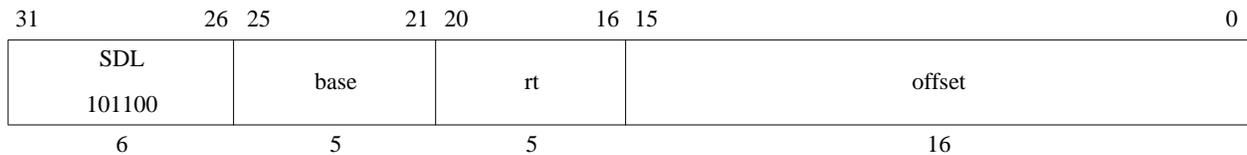
**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 0^3 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

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

## Store Doubleword from Coprocessor 2 — SDC2

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|----|----|----|----|----|----|----|----|
| SDC2<br>111110 | | base | | rt | | offset | |
| 6 | | 5 | | 5 | | 16 | |

**Format:** SDC2 rt, offset(base)                                         **MIPS32**

**Purpose:**

To store a doubleword from a Coprocessor 2 register to memory

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

The 64-bit doubleword in Coprocessor 2 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:**

An Address Error exception occurs if EffectiveAddress$_{2..0}$ ≠ 0 (not doubleword-aligned).

**Operation:**
```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 0^3 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← CPR[2,rt,0]
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

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

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|
| SDL<br>101100 | | base | | rt | | offset | |
| 6 | | 5 | | 5 | | 16 | |

**Format:** SDL rt, offset(base)                                      **MIPS64 (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 8 consecutive bytes forming a doubleword *(DW)* in memory, starting at an arbitrary byte boundary.

A part of *DW*, the most-significant 1 to 8 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 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 6 bytes, is located in the aligned doubleword containing the most-significant byte at 2. First, SDL stores the 6 most-significant bytes of the source register into these bytes in memory. Next, the complementary SDR instruction stores the remainder of *DW*.

**Figure 3-11 Unaligned Doubleword Store With SDL and SDR**
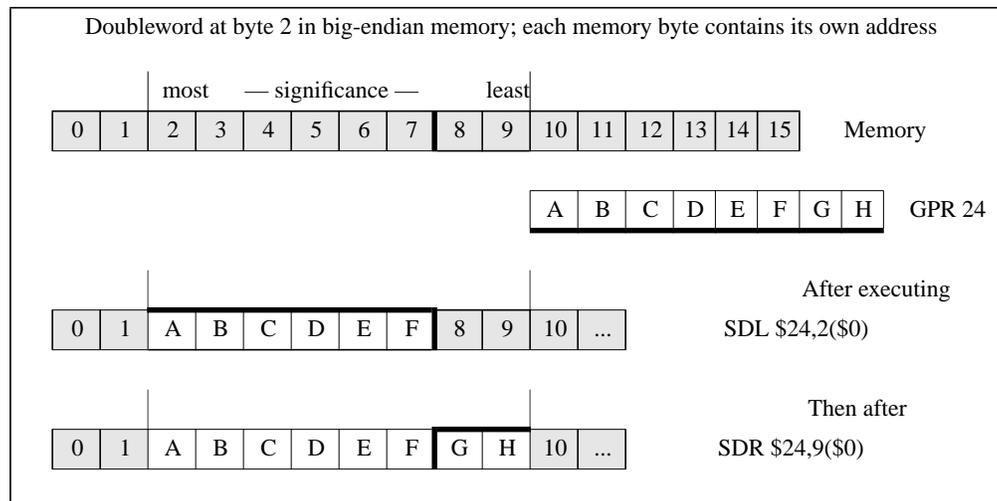
The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned doubleword—that is, the low 3 bits of the address (*vAddr2..0*)—and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes stored for every combination of offset and byte ordering.

**Figure 3-12 Bytes Stored by an SDL Instruction**

| | | | Initial Memory Contents and Byte Offsets | | | | | | | | | Contents of | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| most | | — significance — | | | least | | | | | | | Source Register | | | | | | |
| most | | — significance — | | | | least | | | | | | |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ←big-endian | most | | | — significance — | | | | | least |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i | j | k | l | m | n | o | p | | A | B | C | D | E | F | G | H |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | ←little-endian offset |

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:**

**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← 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
bytesel← vAddr_2..0 xor BigEndianCPU^3
datadoubleword← 0^56-8*bytesel || GPR[rt]_63..56-8*bytesel
StoreMemory (CCA, byte, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Reserved Instruction

```
 31            26  25           21  20          16  15                                    0
┌──────────────┬───────────────┬──────────────┬────────────────────────────────────────┐
│     SDR      │               │              │                                          │
│              │     base      │      rt      │                  offset                  │
│    101101    │               │              │                                          │
└──────────────┴───────────────┴──────────────┴────────────────────────────────────────┘
       6              5              5                           16
```
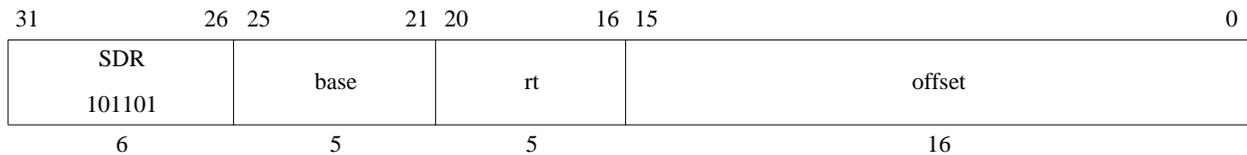
**Format:**   SDR rt, offset(base)                                       **MIPS64 (MIPS III)**

**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 8 consecutive bytes forming a doubleword *(DW)* in memory, starting at an arbitrary byte boundary.

A part of *DW*, the least-significant 1 to 8 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 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 2 bytes, is located in the aligned doubleword containing the least-significant byte at 9. First, SDR stores the 2 least-significant bytes of the source register into these bytes in memory. Next, the complementary SDL stores the remainder of *DW*.

**Figure 3-13 Unaligned Doubleword Store With SDR and SDL**
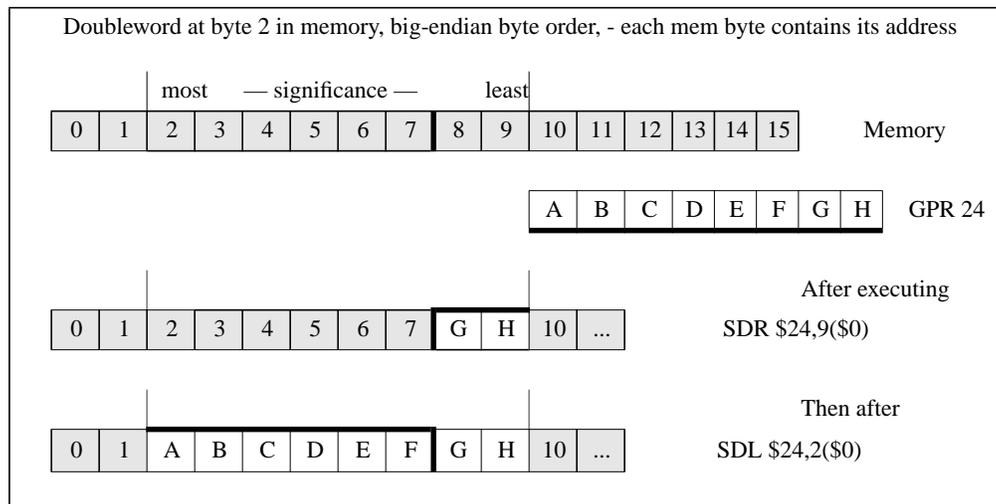
The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned doubleword—that is, the low 3 bits of the address (*vAddr2..0*)—and the current byte ordering mode of the processor (big- or little-endian). Figure 3-14 shows the bytes stored for every combination of offset and byte-ordering.

**Figure 3-14 Bytes Stored by an SDR Instruction**



**Restrictions:**

**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0  xor  ReverseEndian³)
If BigEndianMem = 0 then
    pAddr← pAddr_PSIZE-1..3 || 0³
endif
bytesel← vAddr_1..0 xor BigEndianCPU³
datadoubleword← GPR[rt]_63-8*bytesel || 0^8*bytesel
StoreMemory (CCA, DOUBLEWORD-byte, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Reserved Instruction

**Store Doubleword Indexed from Floating Point**                                                    **SDXC1**

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COP1X 010011 | | base | | index | | fs | | 0 00000 | | SDXC1 001001 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |
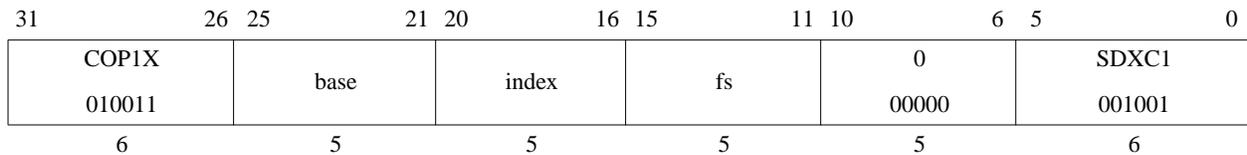
**Format:** SDXC1 fs, index(base)                                                    **MIPS64 (MIPS IV)**

**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.

**Restrictions:**

An Address Error exception occurs if EffectiveAddress$_{2..0}$ ≠ 0 (not doubleword-aligned).
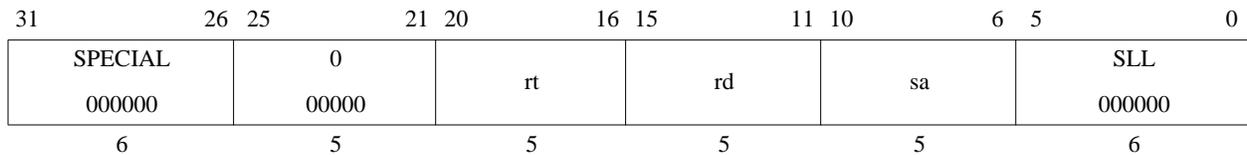
**Operation:**

```
vAddr ← GPR[base] + GPR[index]
if vAddr_2..0 ≠ 0^3 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

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

| 31 | 26 25 | 21 20 | 16 15 | 0 |
|---|---|---|---|---|
| SH<br>101001 | base | rt | offset | |
| 6 | 5 | 5 | 16 | |
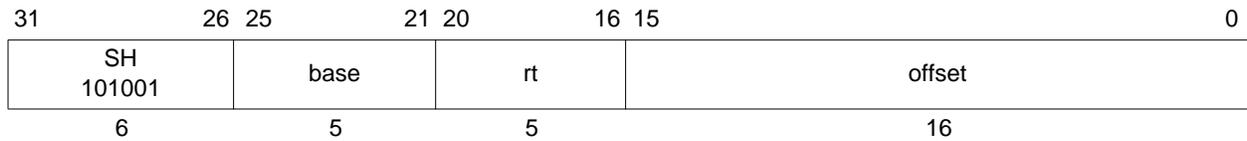
**Format:** `SH rt, offset(base)`      **MIPS32 (MIPS I)**

**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.

**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr₀ ≠ 0 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr1_2..0 xor (ReverseEndian² || 0))
bytesel← vAddr1_2..0 xor (BigEndianCPU² || 0)
datadoubleword← GPR[rt]_63-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, HALFWORD, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Address Error

| 31          26 | 25          21 | 20          16 | 15          11 | 10          6 | 5          0 |
|---|---|---|---|---|---|
| SPECIAL | 0 | | | | SLL |
| 000000 | 00000 | rt | rd | sa | 000000 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**  SLL rd, rt, sa                                                                        **MIPS32 (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 zeros into the emptied bits; the word result is sign-extended and placed in GPR *rd*. The bit-shift amount is specified by *sa.*

**Restrictions:**

None

**Operation:**

```
s      ← sa
temp   ← GPR[rt](31-s)..0 || 0ˢ
GPR[rd]← sign_extend(temp)
```

**Exceptions:**

None

**Programming Notes:**

Unlike nearly all other word operations, the SLL 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.

SLL r0, r0, 0, expressed as NOP, is the assembly idiom used to denote no operation.

SLL r0, r0, 1, expressed as SSNOP, is the assembly idiom used to denote no operation that causes an issue break on superscalar processors.

**Shift Word Left Logical Variable**                                                                          **SLLV**

| 31           26 | 25          21 | 20          16 | 15          11 | 10           6 | 5            0 |
|-----------------|----------------|----------------|----------------|----------------|----------------|
| SPECIAL         | rs             | rt             | rd             | 0              | SLLV           |
| 000000          |                |                |                | 00000          | 000100         |
| 6               | 5              | 5              | 5              | 5              | 6              |

**Format:** SLLV rd, rt, rs                                                                        **MIPS32 (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 zeros into the emptied bits; the result word is sign-extended and placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

**Restrictions: None**

**Operation:**
```
s      ← GPR[rs]₄..₀
temp   ← GPR[rt]₍₃₁₋ₛ₎..₀ || 0ˢ
GPR[rd]← sign_extend(temp)
```

$$s \leftarrow GPR[rs]_{4..0}$$
$$temp \leftarrow GPR[rt]_{(31-s)..0} \| 0^s$$
$$GPR[rd] \leftarrow sign\_extend(temp)$$

**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.

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SPECIAL 000000 | | rs | | rt | | rd | | 0 00000 | | SLT 101010 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:** SLT rd, rs, rt                                           **MIPS32 (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, it is 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

| 31          26 | 25          21 | 20        16 | 15                                              0 |
|----------------|----------------|--------------|---------------------------------------------------|
| SLTI<br>001010 | rs | rt | immediate |
| 6 | 5 | 5 | 16 |
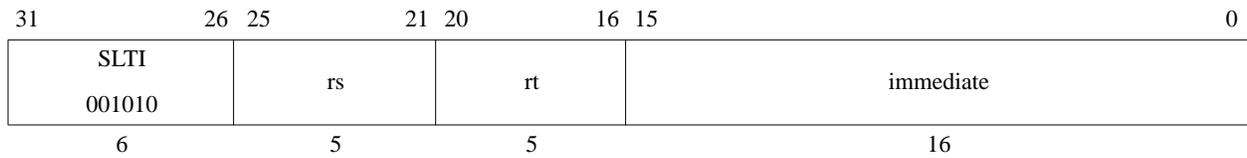
**Format:**  SLTI rt, rs, immediate                                                    **MIPS32 (MIPS I)**

**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, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.
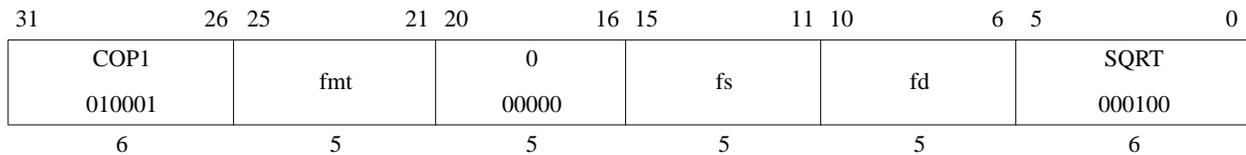
**Restrictions:**

None

**Operation:**

```
if GPR[rs] < sign_extend(immediate) then
    GPR[rd] ← 0^GPRLEN-1|| 1
else
    GPR[rd] ← 0^GPRLEN
endif
```

**Exceptions:**

None

| 31          26 | 25          21 | 20          16 | 15                                    0 |
|----------------|----------------|----------------|------------------------------------------|
| SLTIU 001011   | rs             | rt             | immediate                                |
| 6              | 5              | 5              | 16                                       |
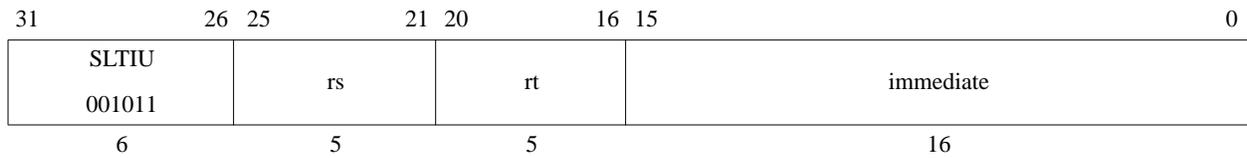
**Format:** SLTIU rt, rs, immediate                                                          **MIPS32 (MIPS I)**

**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, it is 0 (false).

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can 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

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SPECIAL<br>000000 | | rs | | rt | | rd | | 0<br>00000 | | SLTU<br>101011 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:** SLTU rd, rs, rt **MIPS32 (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, it is 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 |
|----|----|----|----|----|----|----|----|----|---|---|---|
| COP1 010001 | | fmt | | 0 00000 | | fs | | fd | | SQRT 000100 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:**  SQRT.S fd, fs                                                                          **MIPS32 (MIPS II)**
            SQRT.D fd, fs                                                                          **MIPS32 (MIPS II)**

**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 is – 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*; if they are not valid, the result is **UNPRE-DICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

**Operation:**

```
StoreFPR(fd, fmt, SquareRoot(ValueFPR(fs, fmt)))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Inexact, Unimplemented Operation

| 31        26 | 25        21 | 20        16 | 15        11 | 10        6 | 5        0 |
|---|---|---|---|---|---|
| SPECIAL<br>000000 | 0<br>00000 | rt | rd | sa | SRA<br>000011 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** SRA rd, rt, sa                                            **MIPS32 (MIPS I)**

**Purpose:**

To execute an arithmetic right-shift of 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 sign-extended and placed in GPR *rd*. The bit-shift amount is specified by *sa*.

**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 UNPREDICTABLE.

**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**

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| SPECIAL 000000 | rs | rt | rd | 0 00000 | SRAV 000111 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** SRAV rd, rt, rs **MIPS32 (MIPS I)**

**Purpose:**

To execute an arithmetic right-shift of 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 sign-extended and placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

**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 UNPREDICTABLE.

**Operation:**

```
if NotWordValue(GPR[rt]) then
    UndefinedResult()
endif
s      ← GPR[rs]₄..₀
temp   ← (GPR[rt]₃₁)ˢ || GPR[rt]₃₁..ₛ
GPR[rd]← sign_extend(temp)
```

**Exceptions:**

None

## Shift Word Right Logical

**SRL**

| 31        | 26 | 25    | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5     | 0 |
|-----------|----|-------|----|----|----|----|----|----|---|-------|---|
| SPECIAL   |    | 0     |    | rt |    | rd |    | sa |   | SRL   |   |
| 000000    |    | 00000 |    |    |    |    |    |    |   | 000010 |  |
| 6         |    | 5     |    | 5  |    | 5  |    | 5  |   | 6     |   |

**Format:** `SRL rd, rt, sa`                                                  **MIPS32 (MIPS I)**

### Purpose:

To execute a logical right-shift of 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 sign-extended and placed in GPR *rd*. The bit-shift amount is specified by *sa.*

### 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 UNPREDICTABLE.

### Operation:

```
if NotWordValue(GPR[rt]) then
    UndefinedResult()
endif
s      ← sa
temp   ← 0^s || GPR[rt]_31..s
GPR[rd]← sign_extend(temp)
```

### Exceptions:

None

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| SPECIAL<br>000000 | rs | rt | rd | 0<br>00000 | SRLV<br>000110 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:** SRLV rd, rt, rs **MIPS32 (MIPS I)**

**Purpose:**

To execute a logical right-shift of 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 sign-extended and placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

**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 UNPREDICTABLE.

**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

| 31        26 | 25       21 | 20       16 | 15       11 | 10      6 | 5       0 |
|:---:|:---:|:---:|:---:|:---:|:---:|
| SPECIAL | 0 | 0 | 0 | 1 | SLL |
| 000000 | 00000 | 00000 | 00000 | 00001 | 000000 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**   SSNOP                                                      **MIPS32**

**Purpose:**

Break superscalar issue on a superscalar processor.

**Description:**

SSNOP is the assembly idiom used to denote superscalar no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 1.

This instruction alters the instruction issue behavior on a superscalar processor by forcing the SSNOP instruction to single-issue. The processor must then end the current instruction issue between the instruction previous to the SSNOP and the SSNOP. The SSNOP then issues alone in the next issue slot.

On a single-issue processor, this instruction is a NOP that takes an issue slot.

**Restrictions:**

None

**Operation:**

None

**Exceptions:**

None

**Programming Notes:**

SSNOP is intended for use primarily to allow the programmer control over CP0 hazards by converting instructions into cycles in a superscalar processor. For example, to insert at least two cycles between an MTC0 and an ERET, one would use the following sequence:

```
mtc0   x,y
ssnop
ssnop
eret
```

Based on the normal issues rules of the processor, the MTC0 issues in cycle T. Because the SSNOP instructions must issue alone, they may issue no earlier than cycle T+1 and cycle T+2, respectively. Finally, the ERET issues no earlier than cycle T+3.  Note that although the instruction after an SSNOP may issue no earlier than the cycle after the SSNOP is issued, that instruction may issue later. This is because other implementation-dependent issue rules may apply that prevent an issue in the next cycle. Processors should not introduce any unnecessary delay in issuing SSNOP instructions.

**Subtract Word**                                                                   **SUB**

| 31          26 | 25        21 | 20        16 | 15        11 | 10         6 | 5          0 |
|----------------|--------------|--------------|--------------|--------------|--------------|
| SPECIAL        | rs           | rt           | rd           | 0            | SUB          |
| 000000         |              |              |              | 00000        | 100010       |
| 6              | 5            | 5            | 5            | 5            | 6            |

**Format:** SUB rd, rs, rt                                                    **MIPS32 (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 sign-extended and placed into GPR *rd*.

**Restrictions:**


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

**Operation:**

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UndefinedResult()
endif
temp ← (GPR[rs]₃₁||GPR[rs]₃₁..₀) − (GPR[rt]₃₁||GPR[rt]₃₁..₀)
if temp₃₂ ≠ temp₃₁ then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← sign_extend(temp₃₁..₀)
endif
```

**Exceptions:**

Integer Overflow

**Programming Notes:**

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

## Floating Point Subtract

**SUB.fmt**

| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|
| COP1 010001 | fmt | ft | fs | fd | SUB 000001 |
| 6 | 5 | 5 | 5 | 5 | 6 |
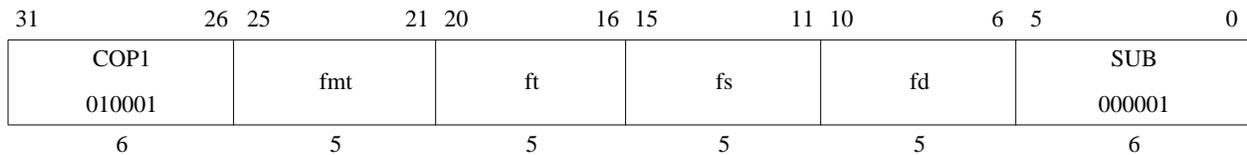
**Format:**  SUB.S fd, fs, ft  **MIPS32 (MIPS I)**
SUB.D fd, fs, ft  **MIPS32 (MIPS I)**
SUB.PS fd, fs, ft  **MIPS64 (MIPS V)**

**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*. SUB.PS subtracts the upper and lower halves of FPR *fs* and FPR *ft* independently, and ORs together any generated exceptional conditions.

**Restrictions:**

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*. If they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of SUB.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

**Operation:**

    StoreFPR (fd, fmt, ValueFPR(fs, fmt) −$_{fmt}$ ValueFPR(ft, fmt))

**CPU Exceptions:**

Coprocessor Unusable, Reserved Instruction

**FPU Exceptions:**

Inexact, Overflow, Underflow, Invalid Op, Unimplemented Op

**Subtract Unsigned Word**                                                                                    **SUBU**

| 31         26 | 25      21 | 20     16 | 15     11 | 10        6 | 5         0 |
|---------------|------------|-----------|-----------|-------------|-------------|
| SPECIAL       | rs         | rt        | rd        | 0           | SUBU        |
| 000000        |            |           |           | 00000       | 100011      |
| 6             | 5          | 5         | 5         | 5           | 6           |

**Format:** SUBU rd, rs, rt                                                                              **MIPS32 (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 sign-extended and placed into GPR *rd*.

No integer overflow exception occurs under any circumstances.

**Restrictions:**

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

**Operation:**

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UndefinedResult()
endif
temp  ← GPR[rs] - GPR[rt]
GPR[rd]← sign_extend(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 unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

## Store Doubleword Indexed Unaligned from Floating Point

**SUXC1**

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COP1X 010011 | | base | | index | | fs | | 0 00000 | | SUXC1 001101 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |
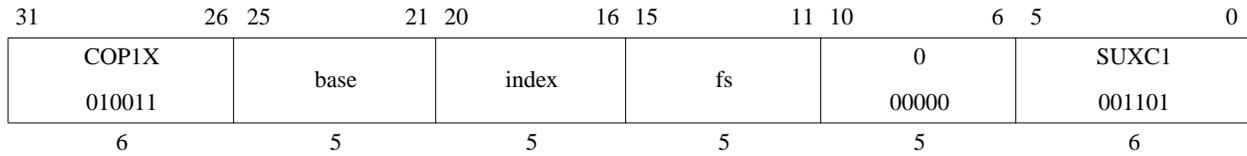
**Format:** SUXC1 fs, index(base)　　　　　　　　　　　　　　　　　**MIPS64 (MIPS V)**

**Purpose:**

To store a doubleword from an FPR to memory (GPR+GPR addressing) ignoring alignment

**Description:** memory[(base+index)$_{PSIZE-1..3}$] ← fs

The contents of the 64-bit doubleword in FPR *fs* is stored at the memory location specified by the effective address. The contents of GPR *index* and GPR *base* are added to form the effective address. The effective address is double-word-aligned; EffectiveAddress$_{2..0}$ are ignored.

**Restrictions:**

The result of this instruction is undefined if the processor is executing in 16 FP registers mode.

**Operation:**

```
vAddr ← (GPR[base]+GPR[index])₆₃..₃ || 0³
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1..3 || (pAddr₂..₀ xor (ReverseEndian || 0²))
bytesel← vAddr₂..₀ xor (BigEndianCPU || 0²)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_WORD) || 0^8*bytesel
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

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

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|
| SW<br>101011 | | base | | rt | | offset | |
| 6 | | 5 | | 5 | | 16 | |

**Format:**  SW rt, offset(base)                                                                          **MIPS32 (MIPS I)**

**Purpose:**

To store a word to memory

**Description:** memory[base+offset] ← 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 2 least-significant bits of the address is non-zero, an Address Error exception occurs.
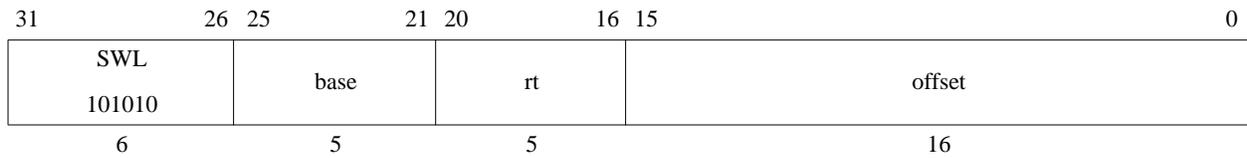
**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr₁..₀ ≠ 0² then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1..3 || (pAddr₂..₀ xor (ReverseEndian || 0²))
bytesel← vAddr₂..₀ xor (BigEndianCPU || 0²)
datadoubleword← GPR[rt]₆₃₋₈*bytesel..₀ || 0^8*bytesel
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Address Error

| 31 | | 26 | 25 | | 21 | 20 | | 16 | 15 | | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| SWC1 111001 | | | base | | | ft | | | offset | | |
| 6 | | | 5 | | | 5 | | | 16 | | |

**Format:** SWC1 ft, offset(base) **MIPS32 (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 $EffectiveAddress_{1..0} \neq 0$ (not word-aligned).

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr₁..₀ ≠ 0³ then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1..3 || (pAddr₂..₀ xor (ReverseEndian || 0²))
bytesel← vAddr₂..₀ xor (BigEndianCPU || 0²)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_WORD) || 0^(8*bytesel)
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

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

| 31 26 | 25 21 | 20 16 | 15 0 |
|---|---|---|---|
| SWC2<br>111010 | base | rt | offset |
| 6 | 5 | 5 | 16 |

**Format:** SWC2 rt, offset(base)                                                                       **MIPS32 (MIPS I)**

**Purpose:**

To store a word from a COP2 register to memory

**Description:** memory[base+offset] ← ft

The low 32-bit word from COP2 (Coprocessor 2) 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:**

An Address Error exception occurs if EffectiveAddress$_{1..0} \neq 0$ (not word-aligned).

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr₂..₀ ≠ 0³ then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1..3 || (pAddr₂..₀ xor (ReverseEndian || 0²))
bytesel ← vAddr₂..₀ xor (BigEndianCPU || 0²)
datadoubleword ← CPR[2,rt,0]₆₃₋₈*bytesel..₀ || 0^8*bytesel
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

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

| 31        26 | 25        21 | 20      16 | 15                              0 |
|:---:|:---:|:---:|:---:|
| SWL<br><br>101010 | base | rt | offset |
| 6 | 5 | 5 | 16 |

**Format:**   SWL rt, offset(base)                                                      **MIPS32 (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 4 consecutive bytes forming a word *(W)* in memory starting at an arbitrary byte boundary.

A part of *W*, the most-significant 1 to 4 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.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWL stores the most-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWR stores the remainder of the unaligned word.

**Figure 3-15 Unaligned Word Store Using SWL and SWR**
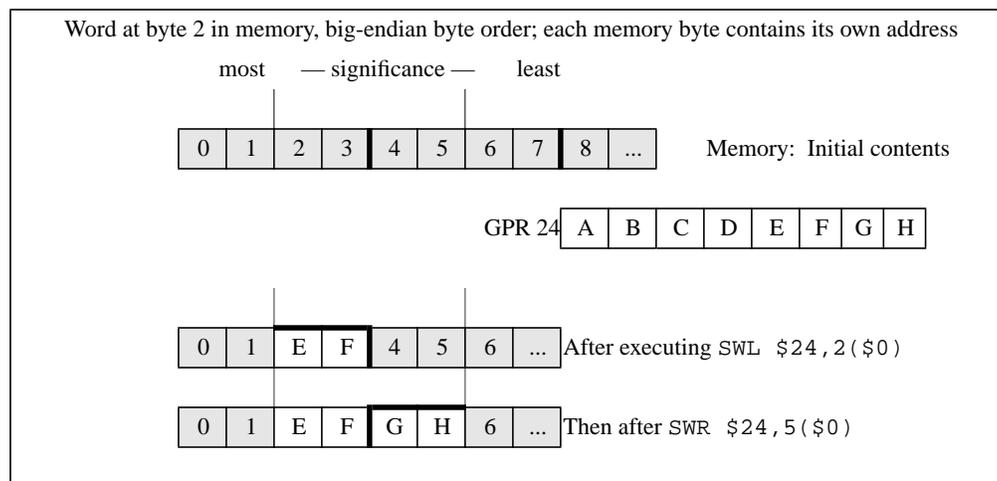


The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address (*vAddr1..0*)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte ordering.

**Figure 3-16 Bytes Stored by an SWL Instruction**
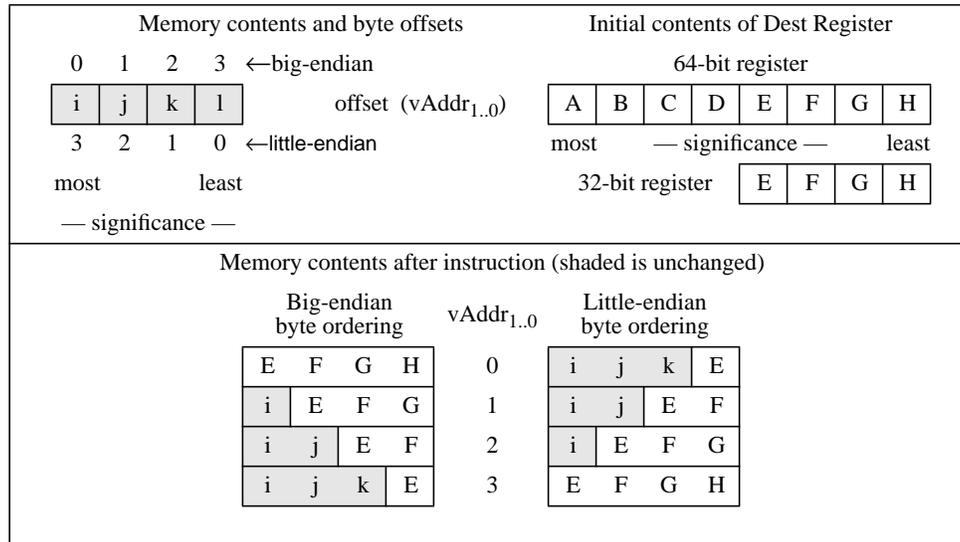


**Restrictions:**

None

**Operation:**

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0  xor  ReverseEndian³)
If BigEndianMem = 0 then
    pAddr ← pAddr_PSIZE-1..2 || 0²
endif
byte   ← vAddr_1..0 xor BigEndianCPU²
if (vAddr_2 xor BigEndianCPU) = 0 then
    datadoubleword ← 0³² || 0²⁴⁻⁸*byte || GPR[rt]_31..24-8*byte
else
    datadoubleword ← 0²⁴⁻⁸*byte || GPR[rt]_31..24-8*byte || 0³²
endif

StoreMemory(CCA, byte, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|---|---|---|---|---|---|---|---|
| SWR 101110 | | base | | rt | | offset | |
| 6 | | 5 | | 5 | | 16 | |
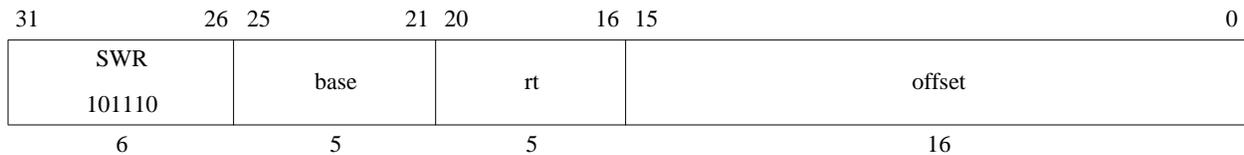
**Format:** SWR rt, offset(base) **MIPS32 (MIPS I)**

**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 4 consecutive bytes forming a word *(W)* in memory starting at an arbitrary byte boundary.

A part of *W*, the least-significant 1 to 4 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.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR stores the least-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWL stores the remainder of the unaligned word.

**Figure 3-17 Unaligned Word Store Using SWR and SWL**
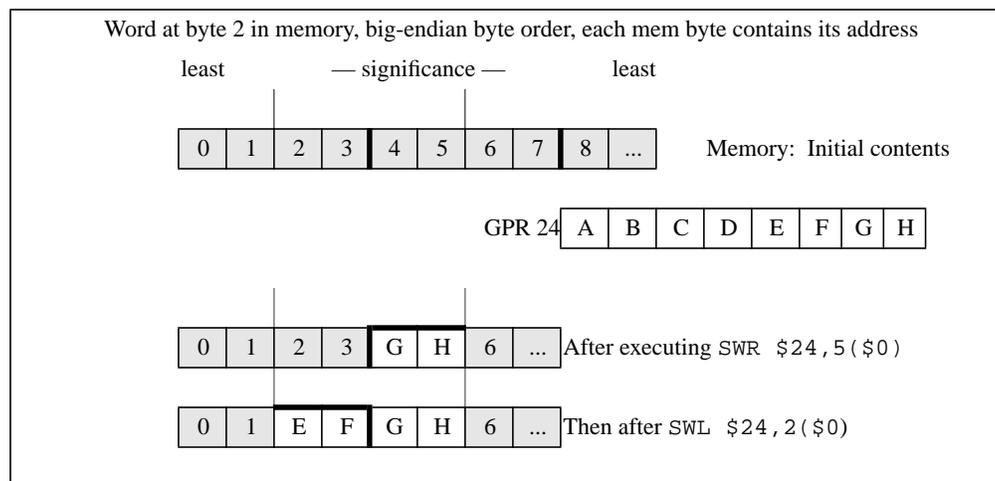
The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address (*vAddr1..0*)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte-ordering.

**Figure 3-18 Bytes Stored by SWR Instruction**
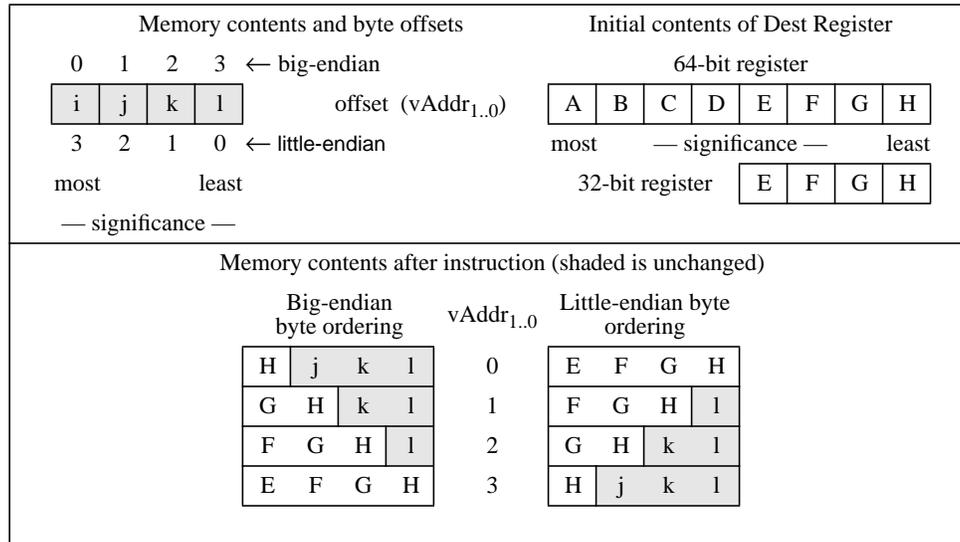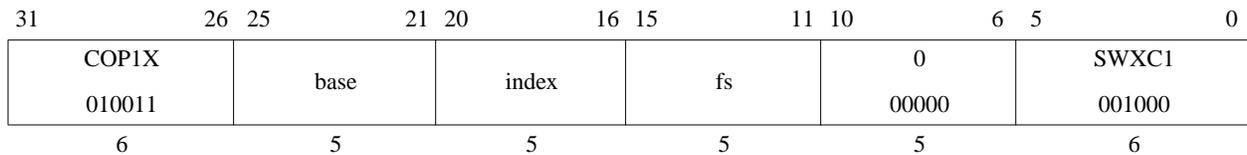


**Restrictions:**

None

**Operation:**
```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor ReverseEndian³)
If BigEndianMem = 0 then
    pAddr ← pAddr_PSIZE-1..2 || 0²
endif
byte   ← vAddr_1..0 xor BigEndianCPU²
if (vAddr_2 xor BigEndianCPU) = 0 then
    datadoubleword ← 0³² || GPR[rt]_31-8*byte..0 || 0^8*byte
else
    datadoubleword ← GPR[rt]_31-8*byte..0 || 0^8*byte || 0³²
endif

StoreMemory(CCA, WORD-byte, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error

**Store Word Indexed from Floating Point** **SWXC1**

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COP1X 010011 | | base | | index | | fs | | 0 00000 | | SWXC1 001000 | |
| 6 | | 5 | | 5 | | 5 | | 5 | | 6 | |

**Format:** SWXC1 fs, index(base) **MIPS64 (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:**

An Address Error exception occurs if EffectiveAddress$_{1..0}$ ≠ 0 (not word-aligned).

**Operation:**

```
vAddr ← GPR[base] + GPR[index]
if vAddr_{1..0} ≠ 0^3 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddr_{PSIZE-1..3} || (pAddr_{2..0} xor (ReverseEndian || 0^2))
bytesel← vAddr_{2..0} xor (BigEndianCPU || 0^2)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_WORD) || 0^{8*bytesel}
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**

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

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| SPECIAL | | 0 | | | | | | stype | | SYNC | |
| 000000 | | 00 0000 0000 0000 0 | | | | | | | | 001111 | |
| 6 | | 15 | | | | | | 5 | | 6 | |

**Format:** SYNC (stype = 0 implied) **MIPS32 (MIPS II)**

**Purpose:**

To order loads and stores.

**Description:**

Simple Description:

- SYNC affects only *uncached* and *cached coherent* loads and stores. The loads and stores that occur before 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.

- SYNC is required, potentially in conjunction with SSNOP, to guarantee that memory reference results are visible across operating mode changes. For example, a SYNC is required on some implementations on entry to and exit from Debug Mode to guarantee that memory affects are handled correctly.

Detailed Description:

- When the *stype* field has a value of zero, every synchronizable load and store that occurs in the instruction stream before the SYNC instruction must be globally performed before any synchronizable load or store that occurs after the SYNC can 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.

-

**Terms:**

*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.

*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 is 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 loads and stores for memory access types other than *uncached* and *cached coherent* is **UNPREDICTABLE**.

**Operation:**
```
SyncOperation(stype)
```

**Exceptions:**

None

**Programming Notes:**

A processor executing load and store instructions observes the order in which loads and stores using the same memory access type occur in the instruction stream; this is known as *program order*.

A *parallel program* has multiple instruction streams that can execute simultaneously 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 store—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 neither is it 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 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 effect of all loads and stores in one group is seen by all processors before the effect of any load or store in the subsequent 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 implementation of 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 generally does not operate on a system that is not strongly ordered. However, a program that does use SYNC works on both types of systems. (System-specific documentation describes the actions needed 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.

SYNC affects the order in which the effects of load and store instructions appear to all processors; it does 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, also is not defined.
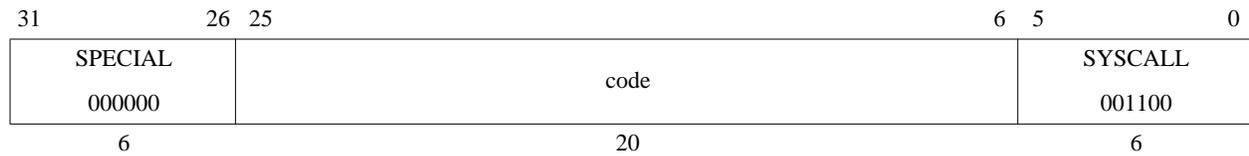
```
# Processor A (writer)
# Conditions at entry:
# The value 0 has been stored in FLAG and that value is observable by B
SW     R1, DATA         # change shared DATA value
LI     R2, 1
SYNC                    # Perform DATA store before performing FLAG store
SW     R2, FLAG         # say that the shared 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 read
      LW     R1, DATA  # Read (valid) shared DATA value
```

Prefetch operations have no effect detectable by User-mode programs, so ordering the effects of prefetch operations is not meaningful.

The code fragments above shows 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.

| 31        26 | 25                          code                          6 | 5           0 |
|--------------|------------------------------------------------------------|---------------|
| SPECIAL<br>000000 | code                                                  | SYSCALL<br>001100 |
| 6 | 20 | 6 |

**Format:** SYSCALL                                                                    **MIPS32 (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.
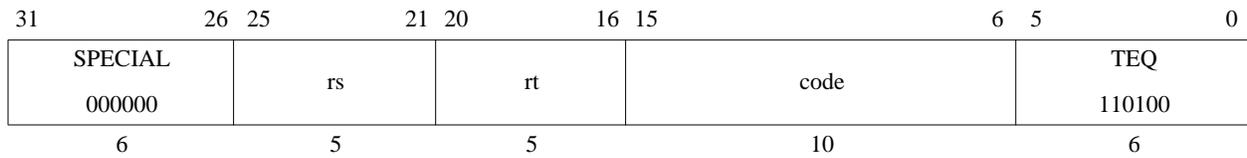
**Restrictions:**

None

**Operation:**

```
SignalException(SystemCall)
```

**Exceptions:**

System Call

| 31 26 | 25 21 | 20 16 | 15 6 | 5 0 |
|---|---|---|---|---|
| SPECIAL<br>000000 | rs | rt | code | TEQ<br>110100 |
| 6 | 5 | 5 | 10 | 6 |

**Format:** `TEQ rs, rt` **MIPS32 (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.
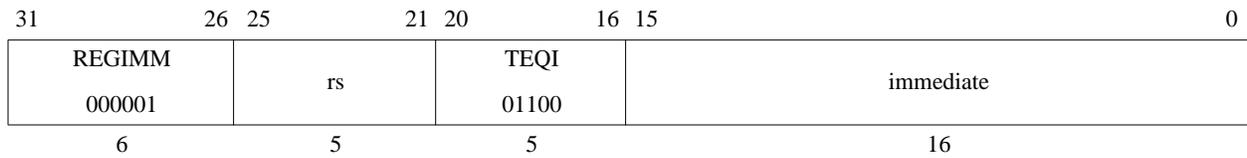
**Restrictions:**

None

**Operation:**

```
if GPR[rs] = GPR[rt] then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|----|----|----|----|----|----|----|---|
| REGIMM<br>000001 | | rs | | TEQI<br>01100 | | immediate | |
| 6 | | 5 | | 5 | | 16 | |

**Format:** `TEQI rs, immediate`                                                            **MIPS32 (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.

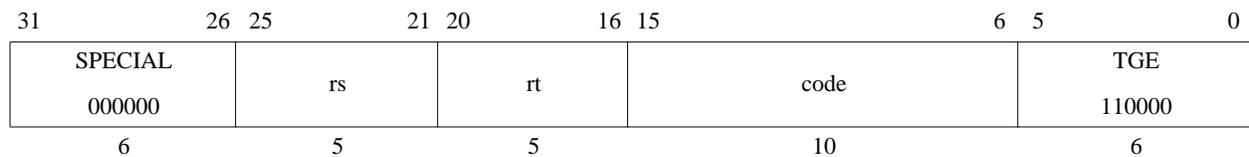**Restrictions:**

None

**Operation:**

```
if GPR[rs] = sign_extend(immediate) then
        SignalException(Trap)
endif
```

**Exceptions:**

Trap

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| SPECIAL<br>000000 | | rs | | rt | | code | | TGE<br>110000 | |
| 6 | | 5 | | 5 | | 10 | | 6 | |

**Format:** TGE rs, rt  **MIPS32 (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.
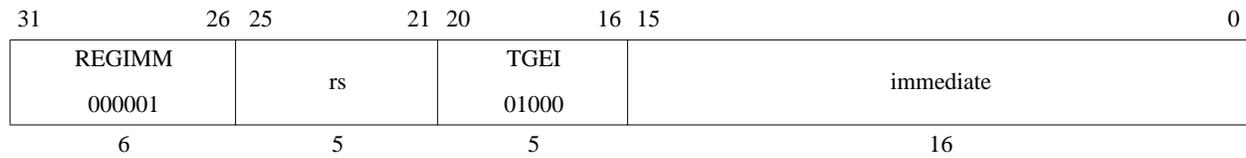
**Restrictions:**

None

**Operation:**

```
if GPR[rs] ≥ GPR[rt] then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

**Trap if Greater or Equal Immediate** <span style="float:right">**TGEI**</span>

| 31 26 | 25 21 | 20 16 | 15 0 |
|---|---|---|---|
| REGIMM<br>000001 | rs | TGEI<br>01000 | immediate |
| 6 | 5 | 5 | 16 |

**Format:** TGEI rs, immediate <span style="float:right">**MIPS32 (MIPS II)**</span>

**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.
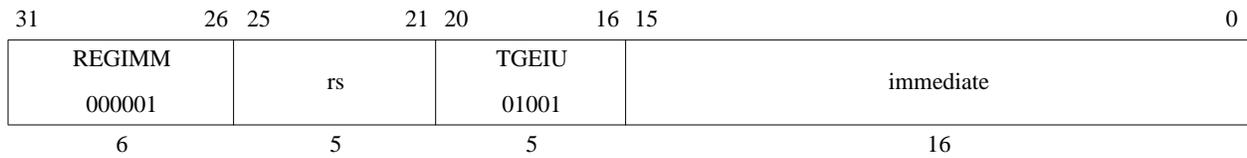
**Restrictions:**

None

**Operation:**

```
if GPR[rs] ≥ sign_extend(immediate) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

| 31          26 | 25          21 | 20          16 | 15                              0 |
|----------------|----------------|----------------|-----------------------------------|
| REGIMM<br>000001 | rs | TGEIU<br>01001 | immediate |
| 6 | 5 | 5 | 16 |

**Format:** `TGEIU rs, immediate`                                                    **MIPS32 (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 can 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.
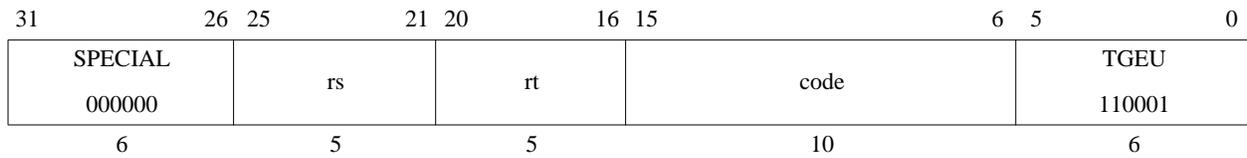
**Restrictions:**

None

**Operation:**

```
if (0 || GPR[rs]) ≥ (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

| 31 26 | 25 21 | 20 16 | 15 6 | 5 0 |
|---|---|---|---|---|
| SPECIAL<br>000000 | rs | rt | code | TGEU<br>110001 |
| 6 | 5 | 5 | 10 | 6 |

**Format:** TGEU rs, rt **MIPS32 (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:**

Trap

**Probe TLB for Matching Entry** **TLBP**

| 31 | 26 | 25 | 24 | | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|
| COP0 | | CO | | 0 | | TLBP | |
| 010000 | | 1 | | 000 0000 0000 0000 0000 | | 001000 | |
| 6 | | 1 | | 19 | | 6 | |

**Format:** TLBP                                                                                              **MIPS32**

**Purpose:**

To find a matching entry in the TLB.

**Description:**

The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the *Index* register is set.

**Restrictions:**


**Operation:**

```
Index ← 1 || UNPREDICTABLE³¹
for i in 0...TLBEntries-1
    if ((TLB[i]_VPN2 and not (TLB[i]_Mask)) =
            (EntryHi_VPN2 and not (TLB[i]_Mask))) and
        (TLB[i]_R = EntryHi_R) and
        ((TLB[i]_G = 1) or (TLB[i]_ASID = EntryHi_ASID)) then
        Index ← i
    endif
endfor
```

**Exceptions:**

Coprocessor Unusable

| 31 | 26 | 25 | 24 | | 6 | 5 | 0 |
|---|---|---|---|---|---|---|---|
| COP0 | | CO | 0 | | | TLBR | |
| 010000 | | 1 | 000 0000 0000 0000 0000 | | | 000001 | |
| 6 | | 1 | 19 | | | 6 | |

**Format:** TLBR **MIPS32**

**Purpose:**

To read an entry from the TLB.

**Description:**

The *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers are loaded with the contents of the TLB entry pointed to by the Index register. Note that the value written to the *EntryHi*, *EntryLo0*, and *EntryLo1* registers may be different from that originally written to the TLB via these registers in that:

- The value returned in the VPN2 field of the *EntryHi* register may havethose bits set to zero corresponding to the one bits in the Mask field of the TLB entry (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed after a TLB entry is written and then read.

- The value returned in the PFN field of the *EntryLo0* and *EntryLo1* registers may havethose bits set to zero corresponding to the one bits in the Mask field of the TLB entry (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed after a TLB entry is written and then read.

- The value returned in the G bit in both the *EntryLo0* and *EntryLo1* registers comes from the single G bit in the TLB entry. Recall that this bit was set from the logical AND of the two G bits in *EntryLo0* and *EntryLo1* when the TLB was written.

**Restrictions:**

The operation is **UNDEFINED** if the contents of the Index register are greater than or equal to the number of TLB entries in the processor.

**Read Indexed TLB Entry**                                                                                   **TLBR**

**Operation:**

```
i ← Index
if i > (TLBEntries - 1) then
    UNDEFINED
endif
PageMask_Mask ← TLB[i]_Mask
EntryHi ← TLB[i]_R || 0^Fill ||
          (TLB[i]_VPN2 and not TLB[i]_Mask) || # Masking implementation dependent
          0^5 || TLB[i]_ASID
EntryLo1 ← 0^Fill ||
          (TLB[i]_PFN1 and not TLB[i]_Mask) || # Masking mplementation dependent
          TLB[i]_C1 || TLB[i]_D1 || TLB[i]_V1 || TLB[i]_G
EntryLo0 ← 0^Fill ||
          (TLB[i]_PFN0 and not TLB[i]_Mask) || # Masking mplementation dependent
          TLB[i]_C0 || TLB[i]_D0 || TLB[i]_V0 || TLB[i]_G
```
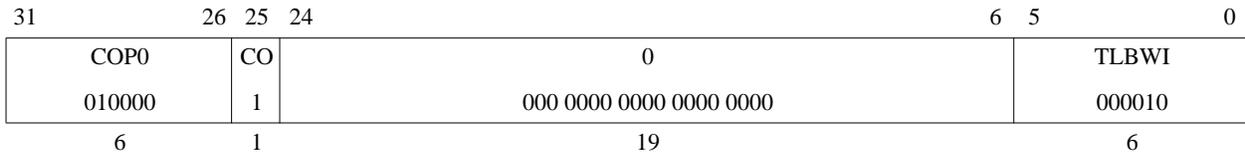
**Exceptions:**

Coprocessor Unusable

| 31 | 26 | 25 | 24 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|
| COP0 | | CO | 0 | | TLBWI | |
| 010000 | | 1 | 000 0000 0000 0000 0000 | | 000010 | |
| 6 | | 1 | 19 | | 6 | |

**Format:** TLBWI **MIPS32**

**Purpose:**

To write a TLB entry indexed by the *Index* register.

**Description:**

The TLB entry pointed to by the Index register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the *PageMask* register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.

- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of *PageMask* register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.

- The single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.

**Restrictions:**

The operation is **UNDEFINED** if the contents of the Index register are greater than or equal to the number of TLB entries in the processor.

**Operation:**

```
i ← Index
TLB[i]_Mask ← PageMask_Mask
TLB[i]_R ← EntryHi_R
TLB[i]_VPN2 ← EntryHi_VPN2 and not PageMask_Mask  # Implementation dependent
TLB[i]_ASID ← EntryHi_ASID
TLB[i]_G ← EntryLo1_G and EntryLo0_G
TLB[i]_PFN1 ← EntryLo1_PFN and not PageMask_Mask  # Implementation dependent
TLB[i]_C1 ← EntryLo1_C
TLB[i]_D1 ← EntryLo1_D
TLB[i]_V1 ← EntryLo1_V
TLB[i]_PFN0 ← EntryLo0_PFN and not PageMask_Mask  # Implementation dependent
TLB[i]_C0 ← EntryLo0_C
TLB[i]_D0 ← EntryLo0_D
TLB[i]_V0 ← EntryLo0_V
```

**Exceptions:**

Coprocessor Unusable

| 31 | 26 | 25 | 24 | 6 | 5 | 0 |
|---|---|---|---|---|---|---|
| COP0 | | CO | 0 | | TLBWR | |
| 010000 | | 1 | 000 0000 0000 0000 0000 | | 000110 | |
| 6 | | 1 | 19 | | 6 | |

**Format:** TLBWR **MIPS32**

**Purpose:**

To write a TLB entry indexed by the *Random* register.

**Description:**

The TLB entry pointed to by the *Random* register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the *PageMask* register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.

- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of *PageMask* register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.

- The value returned in the G bit in both the *EntryLo0* and *EntryLo1* registers comes from the single G bit in the TLB entry. Recall that this bit was set from the logical AND of the two G bits in *EntryLo0* and *EntryLo1* when the TLB was written.
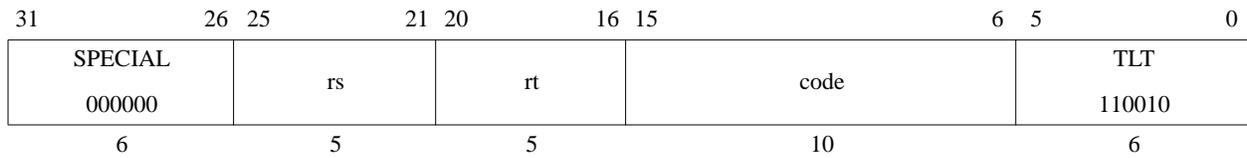
**Restrictions:**

The operation is **UNDEFINED** if the contents of the Index register are greater than or equal to the number of TLB entries in the processor.

**Operation:**

```
i ← Random
TLB[i]_Mask ← PageMask_Mask
TLB[i]_R ← EntryHi_R
TLB[i]_VPN2 ← EntryHi_VPN2 and not PageMask_Mask # Implementation dependent
TLB[i]_ASID ← EntryHi_ASID
TLB[i]_G ← EntryLo1_G and EntryLo0_G
TLB[i]_PFN1 ← EntryLo1_PFN and not PageMask_Mask # Implementation dependent
TLB[i]_C1 ← EntryLo1_C
TLB[i]_D1 ← EntryLo1_D
TLB[i]_V1 ← EntryLo1_V
TLB[i]_PFN0 ← EntryLo0_PFN and not PageMask_Mask # Implementation dependent
TLB[i]_C0 ← EntryLo0_C
TLB[i]_D0 ← EntryLo0_D
TLB[i]_V0 ← EntryLo0_V
```

**Exceptions:**

Coprocessor Unusable

| 31 26 | 25 21 | 20 16 | 15 6 | 5 0 |
|---|---|---|---|---|
| SPECIAL<br>000000 | rs | rt | code | TLT<br>110010 |
| 6 | 5 | 5 | 10 | 6 |

**Format:** `TLT rs, rt` **MIPS32 (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.
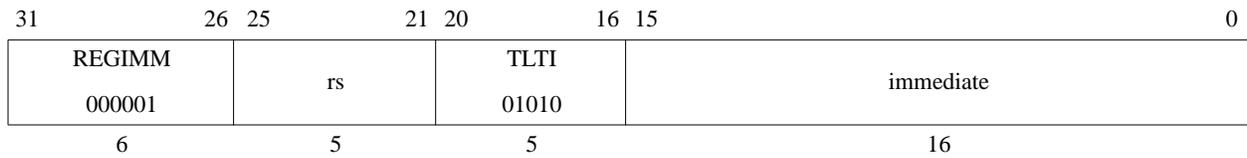
**Restrictions:**

None

**Operation:**

```
if GPR[rs] < GPR[rt] then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

| 31          26 | 25          21 | 20          16 | 15                              0 |
|----------------|----------------|----------------|-----------------------------------|
| REGIMM<br>000001 | rs | TLTI<br>01010 | immediate |
| 6 | 5 | 5 | 16 |

**Format:** `TLTI rs, immediate`                                            **MIPS32 (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.
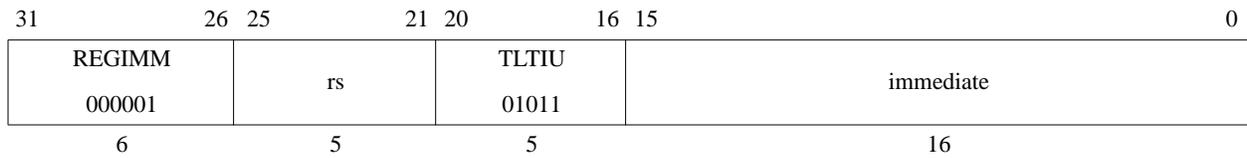
**Restrictions:**

None

**Operation:**

```
if GPR[rs] < sign_extend(immediate) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
|----|----|----|----|----|----|----|----|
| REGIMM<br>000001 | | rs | | TLTIU<br>01011 | | immediate | |
| 6 | | 5 | | 5 | | 16 | |

**Format:** TLTIU rs, immediate                                                                    **MIPS32 (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 can 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.
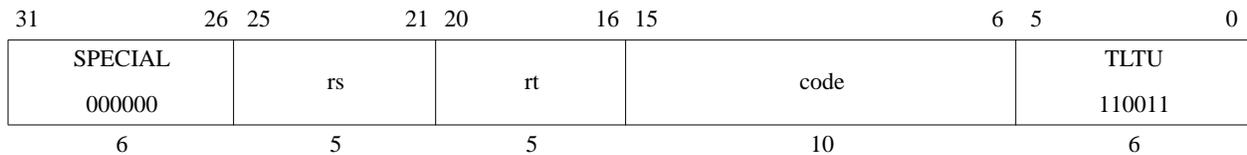
**Restrictions:**

None

**Operation:**

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

## Trap if Less Than Unsigned

**TLTU**

| 31          26 | 25          21 | 20          16 | 15          6 | 5          0 |
|----------------|----------------|----------------|---------------|--------------|
| SPECIAL<br>000000 | rs | rt | code | TLTU<br>110011 |
| 6 | 5 | 5 | 10 | 6 |

**Format:** `TLTU rs, rt`                                           **MIPS32 (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.
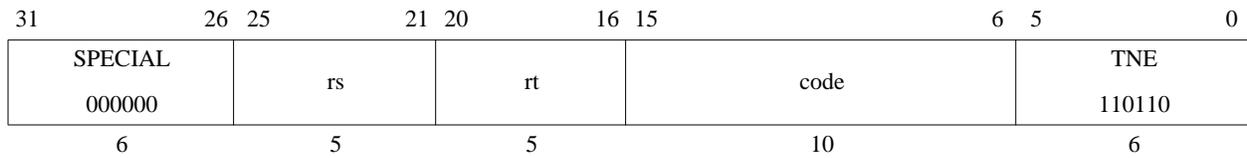
**Restrictions:**

None

**Operation:**

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

| 31 26 | 25 21 | 20 16 | 15 6 | 5 0 |
|---|---|---|---|---|
| SPECIAL<br>000000 | rs | rt | code | TNE<br>110110 |
| 6 | 5 | 5 | 10 | 6 |

**Format:** `TNE rs, rt` **MIPS32 (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.
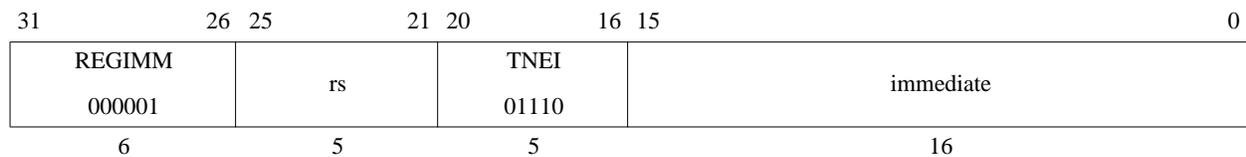
**Restrictions:**

None

**Operation:**

```
if GPR[rs] ≠ GPR[rt] then
        SignalException(Trap)
endif
```

**Exceptions:**

Trap

## Trap if Not Equal

**TNEI**

| REGIMM 000001 | rs | TNEI 01110 | immediate |
|:---:|:---:|:---:|:---:|
| 6 | 5 | 5 | 16 |

31        26  25        21  20        16  15        0

**Format:** TNEI rs, immediate                                   **MIPS32 (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.
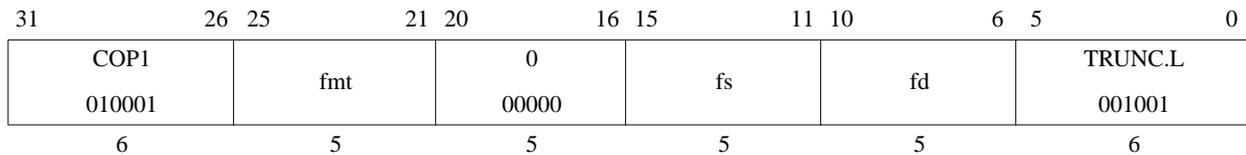
**Restrictions:**

None

**Operation:**

```
if GPR[rs] ≠ sign_extend(immediate) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

| 31         26 | 25        21 | 20        16 | 15        11 | 10        6 | 5         0 |
|---------------|--------------|--------------|--------------|-------------|-------------|
| COP1<br><br>010001 | fmt | 0<br><br>00000 | fs | fd | TRUNC.L<br><br>001001 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**   TRUNC.L.S fd, fs                                      **MIPS64 (MIPS III)**
              TRUNC.L.D fd, fs                                      **MIPS64 (MIPS III)**

**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 and rounded 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. In this case 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*.

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.
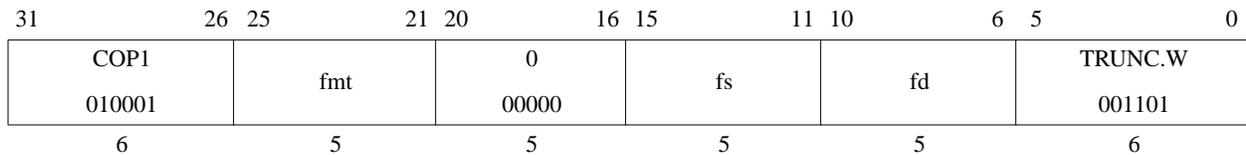
**Operation:**

```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation, Overflow, Inexact

**Floating Point Truncate to Word Fixed Point**

| 31        26 | 25        21 | 20        16 | 15      11 | 10       6 | 5          0 |
|---|---|---|---|---|---|
| COP1<br>010001 | fmt | 0<br>00000 | fs | fd | TRUNC.W<br>001101 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**   TRUNC.W.S fd, fs                                   **MIPS32 (MIPS II)**
               TRUNC.W.D fd, fs                                   **MIPS32 (MIPS II)**

**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. In this case 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*.

**Restrictions:**

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

**Operation:**

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Invalid Operation, Overflow, Unimplemented Operation

| 31            26 | 25   24 | | 6   5           0 |
|---|---|---|---|
| COP0<br>010000 | CO<br>1 | Implementation-Dependent Code | WAIT<br>100000 |
| 6 | 1 | 19 | 6 |

**Format:** `WAIT`                                                           **MIPS32**

**Purpose:**

Wait for Event

**Description:**

The WAIT instruction performs an implementation-dependent operation, usually involving a lower power mode. Software may use bits 24:6 of the instruction to communicate additional information to the processor, and the processor may use this information as control for the lower power mode. A value of zero for bits 24:6 is the default and must be valid in all implementations.

The WAIT instruction is typically implemented by stalling the pipeline at the completion of the instruction and entering a lower power mode. The pipeline is restarted when an external event, such as an interrupt or external request occurs, and execution continues with the instruction following the WAIT instruction. It is implementation-dependent whether the pipeline restarts when a non-enabled interrupt is requested. In this case, software must poll for the cause of the restart. If the pipeline restarts as the result of an enabled interrupt, that interrupt is taken between the WAIT instruction and the following instruction (EPC for the interrupt points at the instruction following the WAIT instruction).

The assertion of any reset or NMI must restart the pipelihne and the corresponding exception myust be taken.

**Restrictions:**

The operation of the processor is **UNDEFINED** if a WAIT instruction is placed in the delay slot of a branch or a jump.

**Operation:**

```
Enter implementation dependent lower power mode
```

**Exceptions:**

Coprocessor Unusable Exception

| 31        26 | 25        21 | 20        16 | 15        11 | 10        6 | 5        0 |
|---|---|---|---|---|---|
| SPECIAL<br>000000 | rs | rt | rd | 0<br>00000 | XOR<br>100110 |
| 6 | 5 | 5 | 5 | 5 | 6 |

**Format:**   XOR rd, rs, rt                                             **MIPS32 (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.*
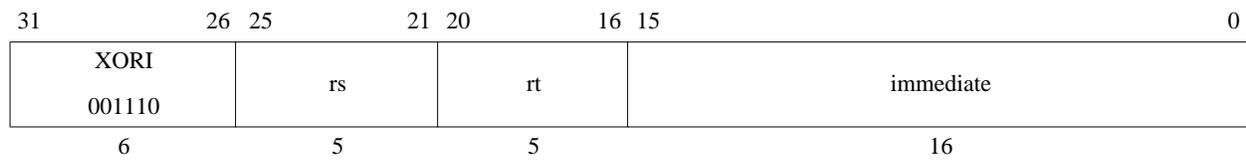
**Restrictions:**

None

**Operation:**

```
GPR[rd] ← GPR[rs] xor GPR[rt]
```

**Exceptions:**

None

| 31          26 | 25          21 | 20          16 | 15                          0 |
|----------------|----------------|----------------|-------------------------------|
| XORI<br>001110 | rs             | rt             | immediate                     |
| 6              | 5              | 5              | 16                            |

**Format:** XORI rt, rs, immediate **MIPS32 (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

# Revision History

| Revision | Date | Description |
|----------|------|-------------|
| 0.90 | November 1, 2000 | Internal review copy of reorganized and updated architecture documentation. |
| 0.91 | November 15, 2000 | External review copy of reorganized and updated architecture documentation. |
| 0.92 | December 15, 2000 | Changes in this revision:<br><br>• Correct sign in description of MSUBU.<br><br>• Update JR and JALR instructions to reflect the changes required by MIPS16. |
| 0.95 | March 12, 2001 | Update for second external review release. |