MIPS64® Architecture For Programmers Volume II: The MIPS64® Instruction Set

Document Number: MD00087
Revision 2.50
July 1, 2005

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

Template: B1.14, Built with tags: 2B ARCH FPU_PS FPU_PSandARCH MIPS64

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 MIPS16e™ 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

1.2.3 UNSTABLE

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

UNSTABLE values have one implementation restriction:

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

1.3 Special Symbols in Pseudocode Notation

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

Table 1-1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
←	Assignment
=, ≠	Tests for equality and inequality
||	Bit string concatenation
x^y	A *y*-bit string formed by *y* copies of the single-bit value *x*
b#n	A constant value *n* in base *b*. For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10.
0bn	A constant value *n* in base *2*. For instance 0b100 represents the binary value 100 (decimal 4).
0xn	A constant value *n* in base *16*. For instance 0x100 represents the hexadecimal value 100 (decimal 256).
$x_{y..z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation (rightmost bit is 0) is used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
+, −	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. In Release 2 of the Architecture, GPR[x] is a short-hand notation for *SGPR[$SRSCtl_{CSS}$, x]*.
SGPR[s,x]	In Release 2 of the Architecture, multiple copies of the CPU general-purpose registers may be implemented. *SGPR[s,x]* refers to GPR set *s*, register *x*.
FPR[x]	Floating Point operand register *x*
FCC[CC]	Floating Point condition code CC. *FCC[0]* has the same value as *COC[1]*.
FPR[x]	Floating Point (Coprocessor unit 1), general register *x*

Table 1-1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
CPR[z,x,s]	Coprocessor unit *z*, general register *x*, select *s*
CP2CPR[x]	Coprocessor unit 2, general register *x*
CCR[z,x]	Coprocessor unit *z*, control register *x*
CP2CCR[x]	Coprocessor unit 2, control register *x*
COC[z]	Coprocessor unit *z* condition signal
Xlat[x]	Translation of the MIPS16e GPR number *x* into the corresponding 32-bit GPR number
BigEndianMem	Endian mode as configured at chip reset (0 →Little-Endian, 1 → Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution.
BigEndianCPU	The endianness for load and store instructions (0 → Little-Endian, 1 → Big-Endian). In User mode, this endianness may be switched by setting the *RE* bit in the *Status* register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).
ReverseEndian	Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as (SR_{RE} and User mode).
LLbit	Bit of **virtual** state used to specify operation for instructions that provide atomic read-modify-write. *LLbit* is set when a linked load occurs and is tested by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.
I:, **I+n**:, **I-n**:	This occurs as a prefix to *Operation* description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to "execute." Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of **I**. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction **I**, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled **I+1**. The effect of pseudocode statements for the current instruction labelled **I+1** appears to occur "at the same time" as the effect of pseudocode statements labeled **I** for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur "at the same time," there is no defined order. Programs must not depend on a particular order of evaluation between such sections.
PC	The *Program Counter* value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to *PC* during an instruction time. If no value is assigned to *PC* during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the *PC* during the instruction time of the instruction in the branch delay slot. In the MIPS Architecture, the PC value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. The PC value contains a full 64-bit address all of which are significant during a memory reference.
ISA Mode	In processors that implement the MIPS16e Application Specific Extension, the *ISA Mode* is a single-bit register that determines in which mode the processor is executing, as follows: **Encoding** / **Meaning** 0 — The processor is executing 32-bit MIPS instructions 1 — The processor is executing MIIPS16e instructions In the MIPS Architecture, the ISA Mode value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the ISA Mode into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception.

Table 1-1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
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. In MIPS32 implementations, **FP32RegistersMode** is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case **FP32RegisterMode** is computed from the FR bit in the *Status* register. If this bit is a 0, the processor operates as if it had 32 32-bit FPRs. If this bit is a 1, the processor operates with 32 64-bit FPRs. The value of **FP32RegistersMode** is computed from the FR bit in the *Status* register.
InstructionInBranchDelaySlot	Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the *dynamic* state of the instruction, not the *static* state. That is, the value is false if a branch or jump occurs to an instruction whose PC immediately follows a branch or jump, but which is not executed in the delay slot of a branch or jump.
SignalException(exception, argument)	Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument). Control does not return from this pseudocode function - the exception is signaled at the point of the call.

1.4 For More Information

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

http://www.mips.com

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

MIPS Architecture Group
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:

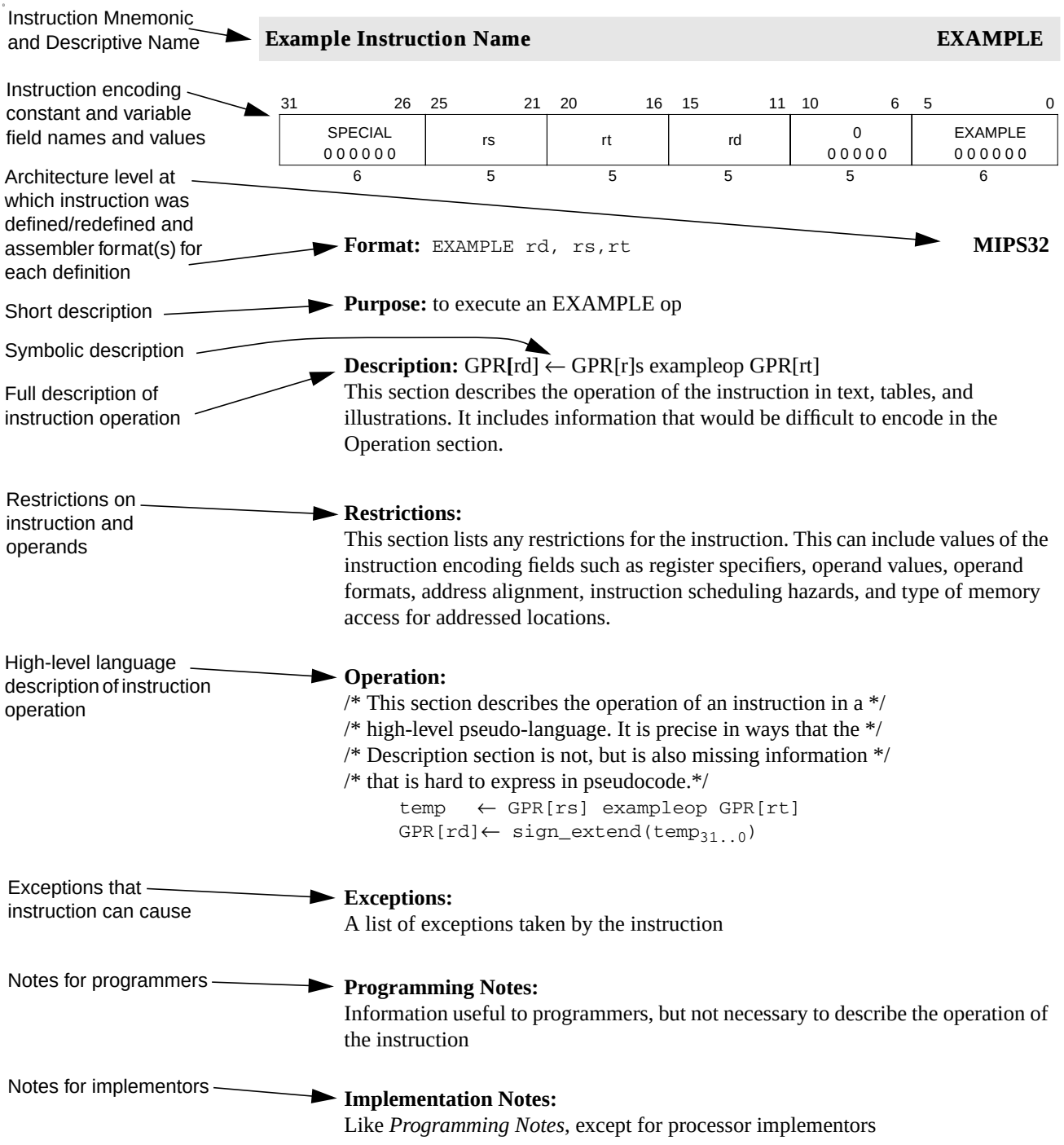


Figure 2-1 Example of Instruction Description

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**.

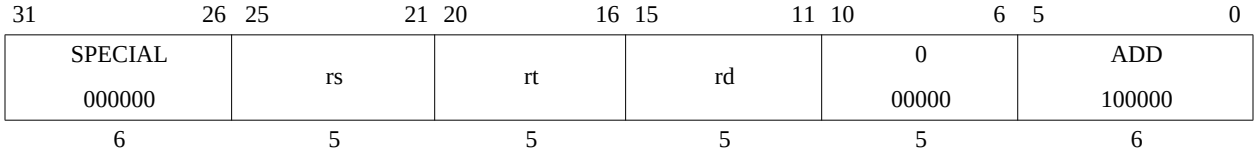


Figure 2-2 Example of Instruction Fields

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**

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.

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: `GPR[rd] ← GPR[rs] + GPR[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:

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:

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

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 2-15 CoprocessorOperation Pseudocode Function

2.2.2.2 Memory Operation 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 2-16 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 2-17 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 2-18 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 2-19 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)

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 2-20 SyncOperation Pseudocode Function

2.2.2.3 Floating Point Functions

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 formattted 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 (fpr0 ≠ 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 2-21 ValueFPR Pseudocode Function

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

```
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] ← UNPREDICTABLE^32 || value_31..0

      D, UNINTERPRETED_DOUBLEWORD:
         if (FP32RegistersMode = 0)
            if (fpr_0 ≠ 0) then
               UNPREDICTABLE
            else
               FPR[fpr]   ← UNPREDICTABLE^32 || value_31..0
               FPR[fpr+1] ← UNPREDICTABLE^32 || 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 2-22 StoreFPR Pseudocode Function

The pseudocode shown below checks for an enabled floating point exception and conditionally signals the exception.

CheckFPException

```
CheckFPException()

/* A floating point exception is signaled if the E bit of the Cause field is a 1 */
/* (Unimplemented Operations have no enable) or if any bit in the Cause field */
/* and the corresponding bit in the Enable field are both 1 */

   if (   (FCSR17 = 1) or
          ((FCSR16..12 and FCSR11..7) ≠ 0))   ) then
      SignalException(FloatingPointException)
   endif

endfunction CheckFPException
```

Figure 2-23 CheckFPException 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 2-24 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 2-25 SetFPConditionCode Pseudocode Function

2.2.2.4 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

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 2-26 SignalException Pseudocode Function

SignalDebugBreakpointException

The SignalDebugBreakpointException function signals a condition that causes entry into Debug Mode from non-Debug Mode.

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

```
SignalDebugBreakpointException()

endfunction SignalDebugBreakpointException
```

Figure 2-27 SignalDebugBreakpointException Pseudocode Function

SignalDebugModeBreakpointException

The SignalDebugModeBreakpointException function signals a condition that causes entry into Debug Mode from Debug Mode (i.e., an exception generated while already running in Debug Mode).

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

```
SignalDebugModeBreakpointException()

endfunction SignalDebugModeBreakpointException
```

Figure 2-28 SignalDebugModeBreakpointException Pseudocode Function

NullifyCurrentInstruction

The NullifyCurrentInstruction function nullifies the current instruction.

The instruction is aborted, inhibiting not only the functional effect of the instruction, but also inhibiting all exceptions detected during fetch, decode, or execution of the instruction in question. For branch-likely instructions, nullification kills the instruction in the delay slot of the branch likely instruction.

```
NullifyCurrentInstruction()

endfunction NullifyCurrentInstruction
```

Figure 2-29 NullifyCurrentInstruction PseudoCode Function

JumpDelaySlot

The JumpDelaySlot function is used in the pseudocode for the PC-relative instructions in the MIPS16e ASE. 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 2-30 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 */

   NotWordValue ← value63..32 ≠ (value31)32

endfunction NotWordValue
```

Figure 2-31 NotWordValue Pseudocode Function

PolyMult

The PolyMult function multiplies two binary polynomial coefficients.

```
PolyMult(x, y)
   temp ← 0
   for i in 0 .. 31
      if xi = 1 then
         temp ← temp xor (y(31-i)..0 || 0i)
      endif
   endfor

   PolyMult ← temp

endfunction PolyMult
```

Figure 2-32 PolyMult 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 MIPS16e instructions.

See Section 2.3, "Op and Function Subfield Notation" on page 22 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.Supersetting of the MIPS64 Architecture is only allowed by adding functions to the *SPECIAL2* major opcode, by adding control for co-processors via the *COP2*, *LWC2*, *SWC2*, *LDC2*, and/or *SDC2*, and/or COP3 opcodes, or via the addition of approved Application Specific Extensions.

The instruction set subsetting rules are as follows:

- All CPU instructions must be implemented - no subsetting is allowed.
- The FPU and related support instructions, including the MOVF and MOVT CPU instructions, may be omitted. Software may determine if an FPU is implemented by checking the state of the FP bit in the *Config1* CP0 register. If the FPU is implemented, the paired single (PS) format is optional. Software may determine which FPU data types are implemented by checking the appropriate bit in the *FIR* CP1 register. The following allowable FPU subsets are compliant with the MIPS64 architecture:
  - No FPU
  - FPU with S, D, W, and L formats and all supporting instructions
  - FPU with S, D, PS, W, and L formats and all supporting instructions
- Coprocessor 2 is optional and may be omitted. Software may determine if Coprocessor 2 is implemented by checking the state of the C2 bit in the *Config1* CP0 register. If Coprocessor 2 is implemented, the Coprocessor 2 interface instructions (BC2, CFC2, COP2, CTC2, DMFC2, DMTC2, LDC2, LWC2, MFC2, MTC2, SDC2, and SWC2) may be omitted on an instruction-by-instruction basis.
- Implementation of the full 64-bit address space is optional. The processor may implement 64-bit data and operations with a 32-bit only address space. In this case, the MMU acts as if 64-bit addressing is always disabled. Software may determine if the processor implements a 32-bit or 64-bit address space by checking the AT field in the *Config* CP0 register.
- Supervisor Mode is optional. If Supervisor Mode is not implemented, bit 3 of the *Status* register must be ignored on write and read as zero.
- The standard TLB-based memory management unit may be replaced with a simpler MMU (e.g., a Fixed Mapping MMU). If this is done, the rest of the interface to the Privileged Resource Architecture must be preserved. If a TLB-based memory management unit is implemented, it must be the standard TLB-based MMU as described in the Privileged Resource Architecture chapter. Software may determine the type of the MMU by checking the MT field in the *Config* CP0 register.
- The Privileged Resource Architecture includes several implementation options and may be subsetted in accordance with those options.
- Instruction, CP0 Register, and CP1 Control Register fields that are marked “Reserved” or shown as “0” in the description of that field are reserved for future use by the architecture and are not available to implementations. Implementations may only use those fields that are explicitly reserved for implementation dependent use.
- Supported ASEs are optional and may be subsetted out. If most cases, software may determine if a supported ASE is implemented by checking the appropriate bit in the *Config1* or *Config3* CP0 register. If they are implemented, they

must implement the entire ISA applicable to the component, or implement subsets that are approved by the ASE specifications.

- EJTAG is optional and may be subsetted out. If it is implemented, it must implement only those subsets that are approved by the EJTAG specification.
- If any instruction is subsetted out based on the rules above, an attempt to execute that instruction must cause the appropriate exception (typically Reserved Instruction or Coprocessor Unusable).

3.2 Alphabetical List of Instructions

Table 3-1 through Table 3-24 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

Table 3-1 CPU Arithmetic Instructions

Mnemonic	Instruction	
MUL	Multiply Word to GPR	
MULT	Multiply Word	
MULTU	Multiply Unsigned Word	
SEB	Sign-Extend Byte	Release 2 Only
SEH	Sign-Extend Halftword	Release 2 Only
SLT	Set on Less Than	
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	
JALR.HB	Jump and Link Register with Hazard Barrier	Release 2 Only
JR	Jump Register	
JR.HB	Jump Register with Hazard Barrier	Release 2 Only

Table 3-3 CPU Instruction Control Instructions

Mnemonic	Instruction	
EHB	Execution Hazard Barrier	Release 2 Only
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	
SYNCI	Synchronize Caches to Make Instruction Writes Effective	Release 2 Only

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 Insert/Extract Instructions

Mnemonic	Instruction	
DEXT	Doubleword Extract Bit Field	Release 2 Only
DEXTM	Doubleword Extract Bit Field Middle	Release 2 Only
DEXTU	Doubleword Extract Bit Field Upper	Release 2 Only
DINS	Doubleword Insert Bit Field	Release 2 Only
DINSM	Doubleword Insert Bit Field Middle	Release 2 Only
DINSU	Doubleword Insert Bit Field Upper	Release 2 Only
DSBH	Doubleword Swap Bytes Within Halfwords	Release 2 Only
DSHD	Doubleword Swap Halfwords Within Doublewords	Release 2 Only
EXT	Extract Bit Field	Release 2 Only
INS	Insert Bit Field	Release 2 Only
WSBH	Word Swap Bytes Within Halfwords	Release 2 Only

Table 3-7 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

Table 3-7 CPU Move Instructions

Mnemonic	Instruction	
MTLO	Move To LO Register	
RDHWR	Read Hardware Register	Release 2 Only

Table 3-8 CPU Shift Instructions

Mnemonic	Instruction	
DROTR	Doubleword Rotate Right	Release 2 Only
DROTR32	Doubleword Rotate Right Plus 32	Release 2 Only
DROTRV	Doubleword Rotate Right Variable	Release 2 Only
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	
ROTR	Rotate Word Right	Release 2 Only
ROTRV	Rotate Word Right Variable	Release 2 Only
SLL	Shift Word Left Logical	
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 3-9 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

Table 3-9 CPU Trap Instructions

Mnemonic	Instruction
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 3-10 Obsolete[1] 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

1. 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 3-11 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

Table 3-11 FPU Arithmetic Instructions

Mnemonic	Instruction
NMSUB.fmt	Floating Point Negative Multiply Subtract
RECIP.fmt	Reciprocal Approximation
RSQRT.fmt	Reciprocal Square Root Approximation
SQRT.fmt	Floating Point Square Root
SUB.fmt	Floating Point Subtract

Table 3-12 FPU Branch Instructions

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

Table 3-13 FPU Compare Instructions

Mnemonic	Instruction
C.cond.fmt	Floating Point Compare

Table 3-14 FPU Convert Instructions

Mnemonic	Instruction	
ALNV.PS	Floating Point Align Variable	64-bit FPU Only
CEIL.L.fmt	Floating Point Ceiling Convert to Long Fixed Point	64-bit FPU Only
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	64-bit FPU Only
CVT.PS.S	Floating Point Convert Pair to Paired Single	64-bit FPU Only
CVT.S.PL	Floating Point Convert Pair Lower to Single Floating Point	64-bit FPU Only
CVT.S.PU	Floating Point Convert Pair Upper to Single Floating Point	64-bit FPU Only
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	64-bit FPU Only
FLOOR.W.fmt	Floating Point Floor Convert to Word Fixed Point	
PLL.PS	Pair Lower Lower	64-bit FPU Only
PLU.PS	Pair Lower Upper	64-bit FPU Only
PUL.PS	Pair Upper Lower	64-bit FPU Only
PUU.PS	Pair Upper Upper	64-bit FPU Only

Table 3-14 FPU Convert Instructions

Mnemonic	Instruction	
ROUND.L.fmt	Floating Point Round to Long Fixed Point	64-bit FPU Only
ROUND.W.fmt	Floating Point Round to Word Fixed Point	
TRUNC.L.fmt	Floating Point Truncate to Long Fixed Point	64-bit FPU Only
TRUNC.W.fmt	Floating Point Truncate to Word Fixed Point	

Table 3-15 FPU Load, Store, and Memory Control Instructions

Mnemonic	Instruction	
LDC1	Load Doubleword to Floating Point	
LDXC1	Load Doubleword Indexed to Floating Point	64-bit FPU Only
LUXC1	Load Doubleword Indexed Unaligned to Floating Point	64-bit FPU Only
LWC1	Load Word to Floating Point	
LWXC1	Load Word Indexed to Floating Point	64-bit FPU Only
PREFX	Prefetch Indexed	
SDC1	Store Doubleword from Floating Point	
SDXC1	Store Doubleword Indexed from Floating Point	64-bit FPU Only
SUXC1	Store Doubleword Indexed Unaligned from Floating Point	64-bit FPU Only
SWC1	Store Word from Floating Point	
SWXC1	Store Word Indexed from Floating Point	64-bit FPU Only

Table 3-16 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	
MFHC1	Move Word from High Half of Floating Point Register	Release 2 Only
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	

Table 3-16 FPU Move Instructions

Mnemonic	Instruction	
MTC1	Move Word to Floating Point	
MTHC1	Move Word to High Half of Floating Point Register	Release 2 Only

Table 3-17 Obsolete[1] FPU Branch Instructions

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

1. 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 3-18 Coprocessor Branch Instructions

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

Table 3-19 Coprocessor Execute Instructions

Mnemonic	Instruction
COP2	Coprocessor Operation to Coprocessor 2

Table 3-20 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 3-21 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

Table 3-21 Coprocessor Move Instructions

Mnemonic	Instruction	
MFHC2	Move Word from High Half of Coprocessor 2 Register	Release 2 Only
MTC2	Move Word to Coprocessor 2	
MTHC2	Move Word to High Half of Coprocessor 2 Register	Release 2 Only

Table 3-22 Obsolete[1] Coprocessor Branch Instructions

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

1. 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.

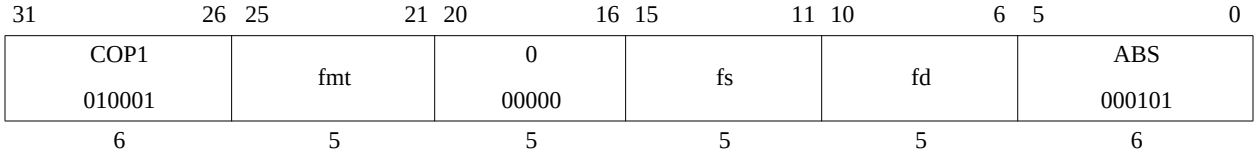
Table 3-23 Privileged Instructions

Mnemonic	Instruction	
CACHE	Perform Cache Operation	
DI	Disable Interrupts	Release 2 Only
DMFC0	Doubleword Move from Coprocessor 0	
DMTC0	Doubleword Move to Coprocessor 0	
EI	Enable Interrupts	Release 2 Only
ERET	Exception Return	
MFC0	Move from Coprocessor 0	
MTC0	Move to Coprocessor 0	
RDPGPR	Read GPR from Previous Shadow Set	Release 2 Only
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	
WRPGPR	Write GPR to Previous Shadow Set	Release 2 Only

Table 3-24 EJTAG Instructions

Mnemonic	Instruction
DERET	Debug Exception Return
SDBBP	Software Debug Breakpoint

Floating Point Absolute Value — ABS.fmt



Format: `ABS.S fd, fs` **MIPS32**

`ABS.D fd, fs` **MIPS32**

`ABS.PS fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

To compute the absolute value of an FP value

Description: `FPR[fd] ← abs(FPR[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:

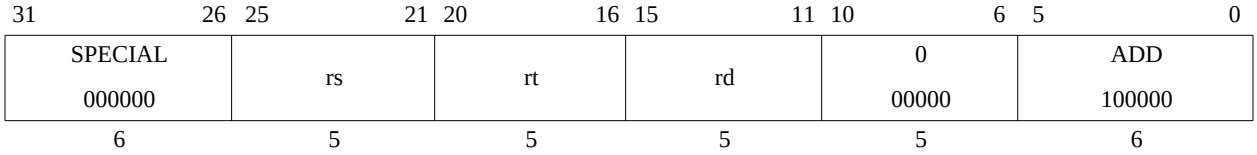
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

Add Word

ADD



Format: `ADD rd, rs, rt` **MIPS32**

Purpose:

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

Description: `GPR[rd] ← GPR[rs] + GPR[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 temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← sign_extend(temp31..0)
endif
```

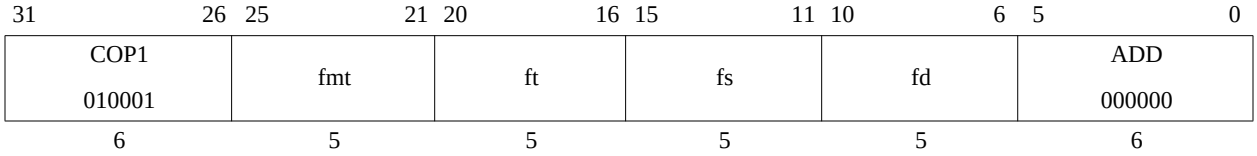
Exceptions:

Integer Overflow

Programming Notes:

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

Floating Point Add — ADD.fmt



Format: `ADD.S fd, fs, ft` **MIPS32**

`ADD.D fd, fs, ft` **MIPS32**

`ADD.PS fd, fs, ft` **MIPS64, MIPS32 Release 2**

Purpose:

To add floating point values

Description: `FPR[fd] ← FPR[fs] + FPR[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))
```

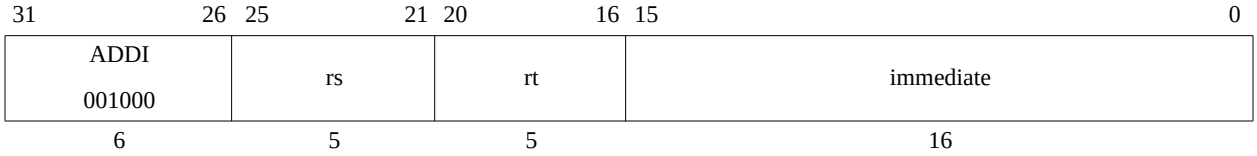
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow

Add Immediate Word — ADDI



Format: `ADDI rt, rs, immediate` **MIPS32**

Purpose:

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

Description: `GPR[rt] ¨ GPR[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
```

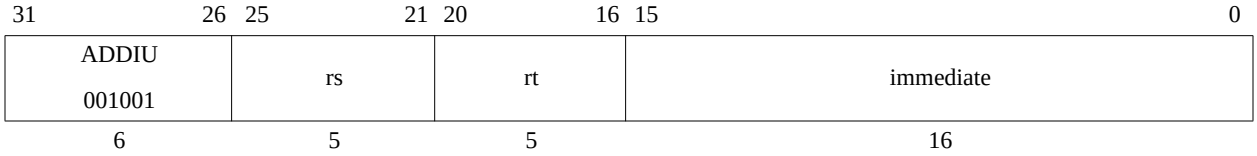
Exceptions:

Integer Overflow

Programming Notes:

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

Add Immediate Unsigned Word — ADDIU



Format: `ADDIU rt, rs, immediate` **MIPS32**

Purpose:

To add a constant to a 32-bit integer

Description: `GPR[rt] ← GPR[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(temp31..0)
```

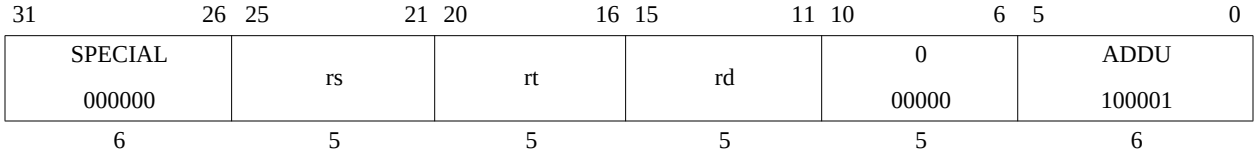
Exceptions:

None

Programming Notes:

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



Format: `ADDU rd, rs, rt` **MIPS32**

Purpose:

To add 32-bit integers

Description: `GPR[rd] ← GPR[rs] + GPR[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:

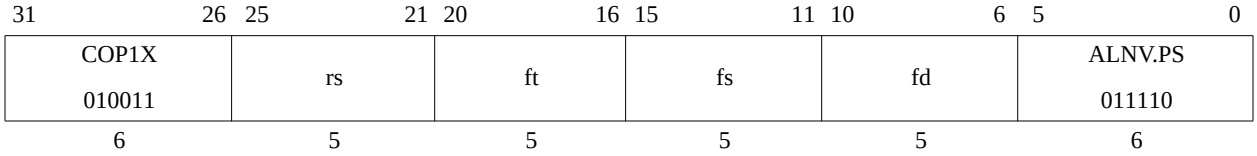
```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← sign_extend(temp31..0)
```

Exceptions:

None

Programming Notes:

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



Format: `ALNV.PS fd, fs, ft, rs` **MIPS64, MIPS32 Release 2**

Purpose:

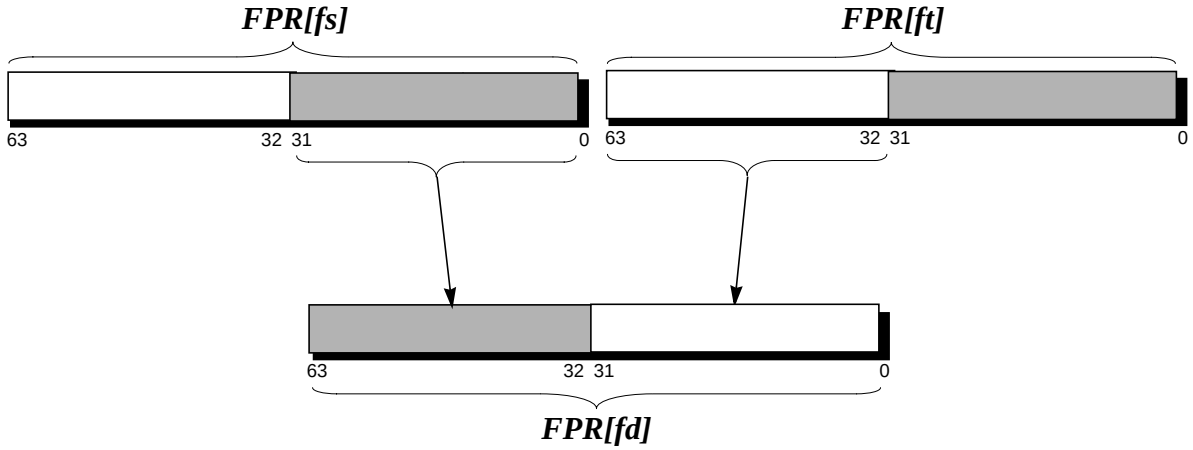
To align a misaligned pair of paired single values

Description: `FPR[fd] ← ByteAlign(GPR[rs]2..0, FPR[fs], FPR[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, FPR *fd* receives FPR *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 FPR *fd* receives the lower half of the paired single value in *fs*, and the lower half of FPR *fd* receives the upper half of the paired single value in FPR *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 **UNPREDICTABLE**.

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:
```

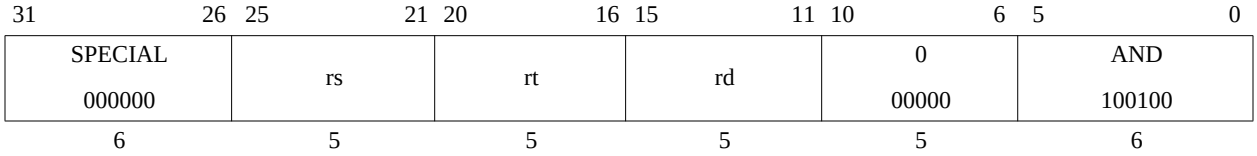
ALNV.PS is also useful with SUXC1 to store paired-single results in a vector loop to a possibly misaligned address:

```
/* 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 */
```



Format: `AND rd, rs, rt` **MIPS32**

Purpose:

To do a bitwise logical AND

Description: `GPR[rd] ← GPR[rs] AND GPR[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

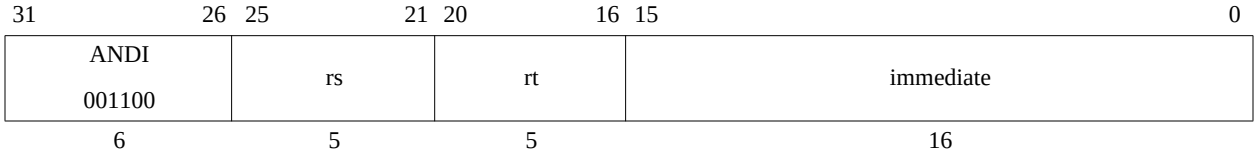
Operation:

```
GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None

And Immediate ANDI



Format: `ANDI rt, rs, immediate` **MIPS32**

Purpose:

To do a bitwise logical AND with a constant

Description: `GPR[rt] ← GPR[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

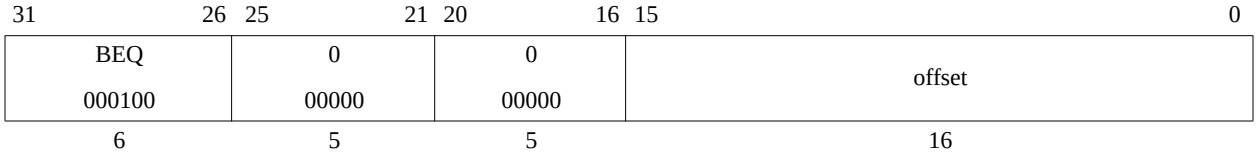
Operation:

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

Exceptions:

None

Unconditional Branch B



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 || 0^2)
I+1:    PC ← PC + target_offset
```

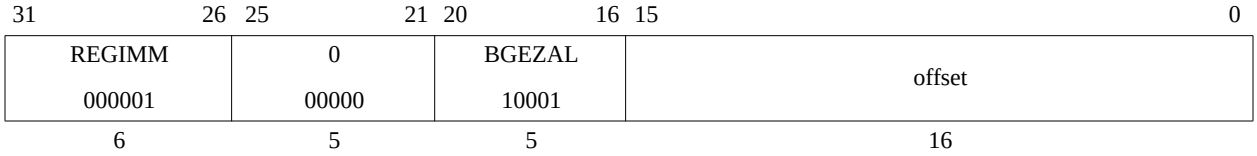
Exceptions:

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.

Branch and Link BAL



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 interpreted 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 re-executed. The result of executing such an instruction is **UNPREDICTABLE**. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

Operation:

```
I:      target_offset ← sign_extend(offset || 0^2)
        GPR[31] ← PC + 8
I+1:    PC ← PC + target_offset
```

Exceptions:

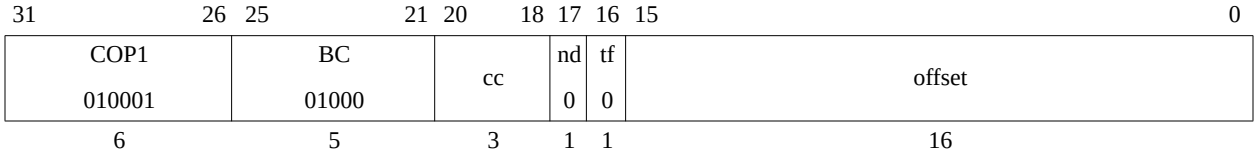
None

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.

Branch on FP False

BC1F



Format: `BC1F offset (cc = 0 implied)` **MIPS32**

`BC1F cc, offset` **MIPS32**

Purpose:

To test an FP condition code and do a PC-relative conditional branch

Description: `if FPConditionCode(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:

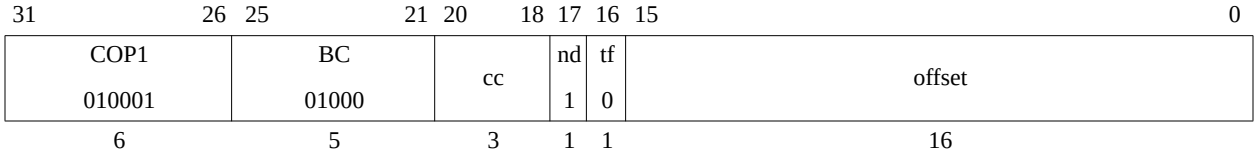
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range

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 **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.



Format: `BC1FL offset (cc = 0 implied)` **MIPS32**

`BC1FL cc, offset` **MIPS32**

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 FPConditionCode(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 ← (offset15)^(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.

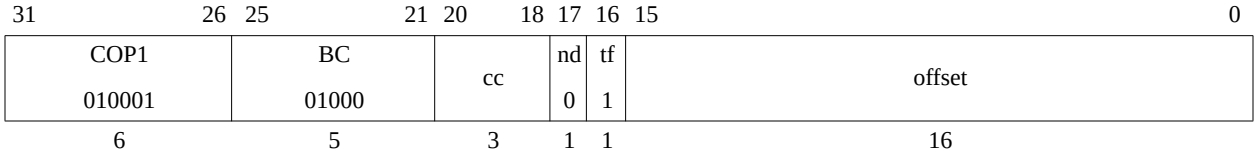
Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability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



Format: `BC1T offset (cc = 0 implied)` **MIPS32**

`BC1T cc, offset` **MIPS32**

Purpose:

To test an FP condition code and do a PC-relative conditional branch

Description: `if FPConditionCode(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 || 0^2
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Programming Notes:

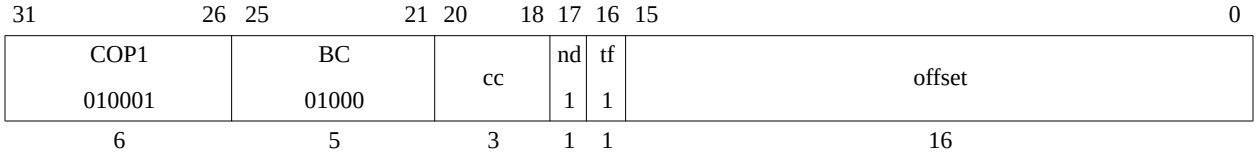
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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.



Format: `BC1TL offset (cc = 0 implied)` **MIPS32**

`BC1TL cc, offset` **MIPS32**

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 FPConditionCode(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
```

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.

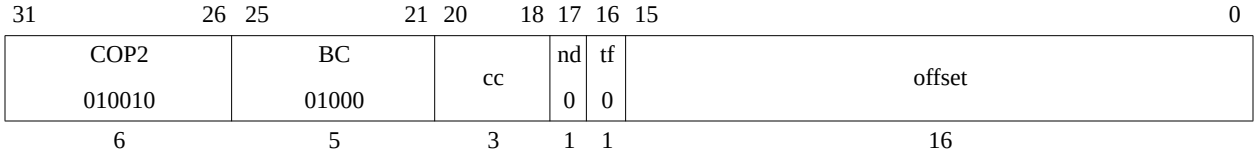
Historical Information:

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.

Branch on COP2 False BC2F



Format: `BC2F offset (cc = 0 implied)` **MIPS32**

`BC2F cc, offset` **MIPS32**

Purpose:

To test a COP2 condition code and do a PC-relative conditional branch

Description: `if COP2Condition(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
```

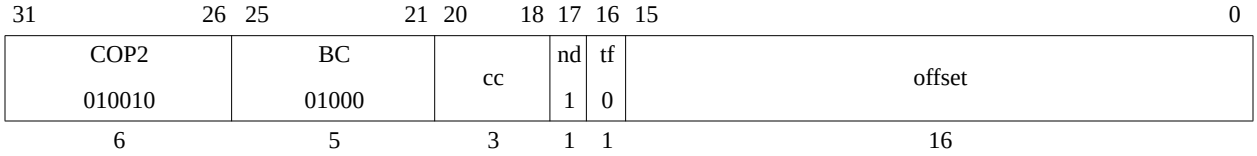
Exceptions:

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.

Branch on COP2 False Likely BC2FL



Format: `BC2FL offset (cc = 0 implied)` **MIPS32**

`BC2FL cc, offset` **MIPS32**

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 COP2Condition(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 || 0^2
I+1:    if condition then
           PC ← PC + target_offset
        else
           NullifyCurrentInstruction()
        endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Programming Notes:

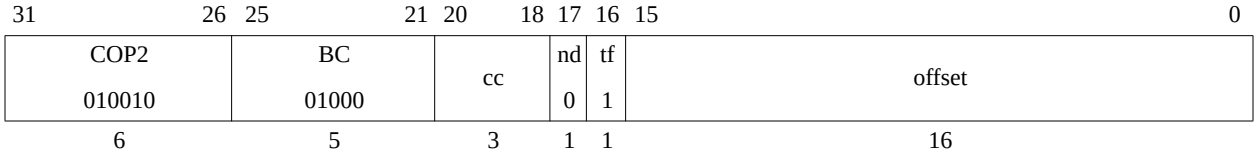
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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.

Branch on COP2 True

BC2T



Format: `BC2T offset (cc = 0 implied)` **MIPS32**

`BC2T cc, offset` **MIPS32**

Purpose:

To test a COP2 condition code and do a PC-relative conditional branch

Description: `if COP2Condition(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 || 0^2
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

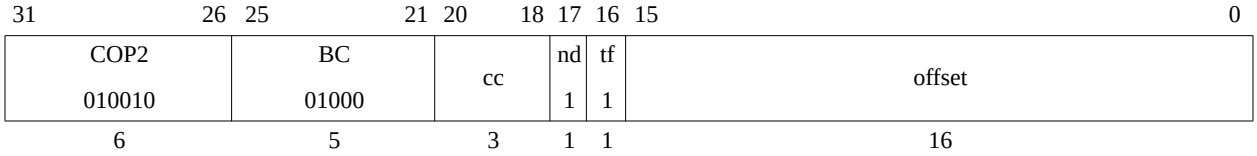
Exceptions:

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.

Branch on COP2 True Likely BC2TL



Format: `BC2TL offset (cc = 0 implied)` **MIPS32**

`BC2TL cc, offset` **MIPS32**

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 COP2Condition(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 || 0^2
I+1:    if condition then
           PC ← PC + target_offset
        else
           NullifyCurrentInstruction()
        endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

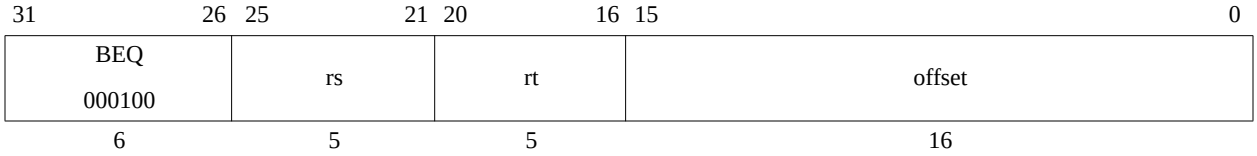
Programming Notes:

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



Format: `BEQ rs, rt, offset` **MIPS32**

Purpose:

To compare GPRs then do a PC-relative conditional branch

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

Exceptions:

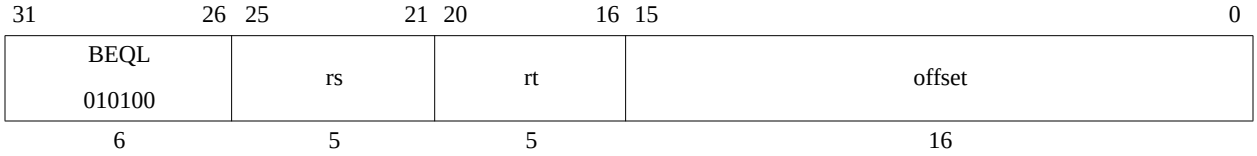
None

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.

Branch on Equal Likely — BEQL



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

Purpose:

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

Description: `if GPR[rs] = GPR[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 || 0^2)
           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.

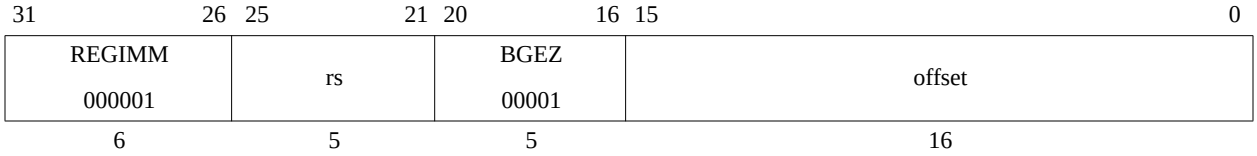
Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

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.

Branch on Greater Than or Equal to Zero — BGEZ



Format: `BGEZ rs, offset` **MIPS32**

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if GPR[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 || 0^2)
        condition ← GPR[rs] ≥ 0^GPRLEN
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

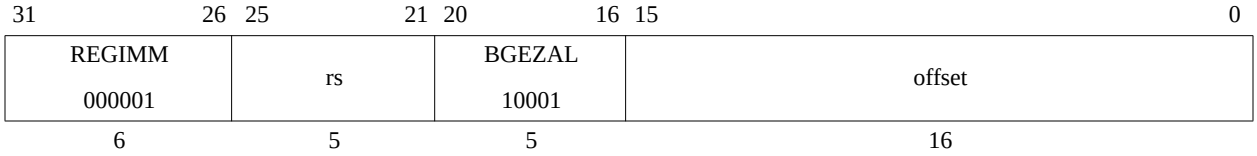
Exceptions:

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



Format: `BGEZAL rs, offset` **MIPS32**

Purpose:

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

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

Exceptions:

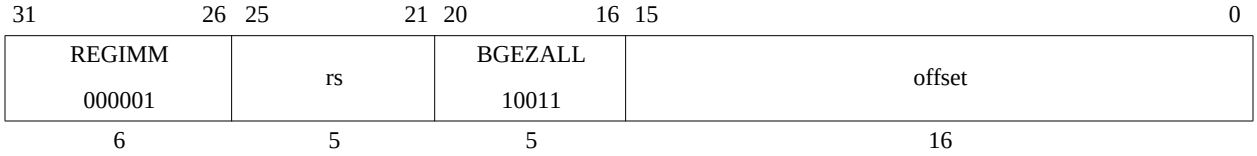
None

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.

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.

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



Format: `BGEZALL rs, offset` **MIPS32**

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 GPR[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 || 0^2)
        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.

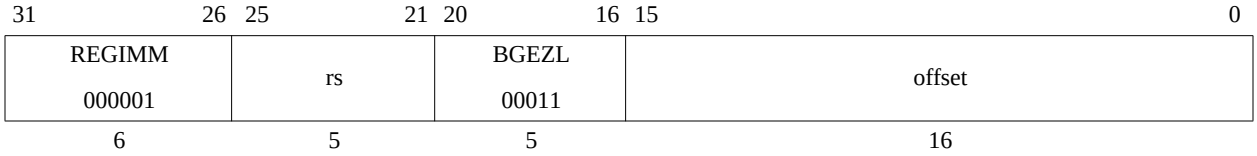
Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

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



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

Purpose:

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

Description: `if GPR[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 || 0^2)
        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.

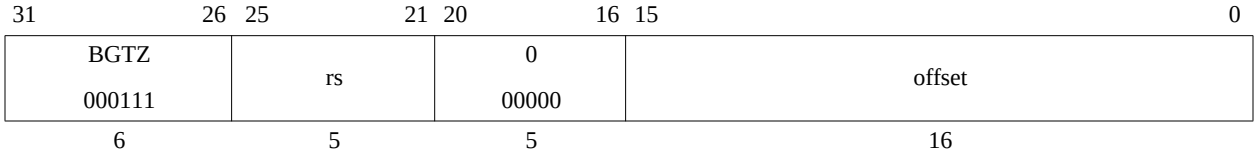
Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

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



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

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if GPR[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 || 0^2)
        condition ← GPR[rs] > 0^GPRLEN
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

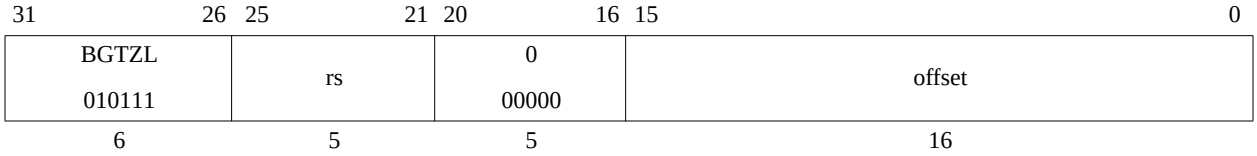
Exceptions:

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 Zero Likely — BGTZL



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

Purpose:

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

Description: `if GPR[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 || 0^2)
        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.

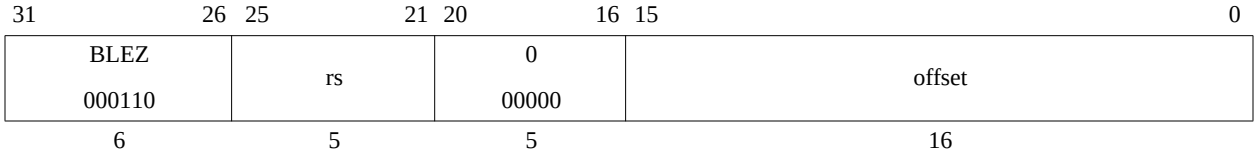
Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

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



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

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if GPR[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 || 0^2)
        condition ← GPR[rs] ≤ 0^GPRLEN
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

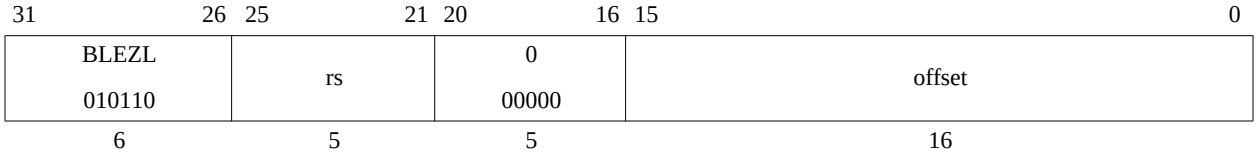
Exceptions:

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 Less Than or Equal to Zero Likely — BLEZL



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

Purpose:

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

Description: `if GPR[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 || 0^2)
        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.

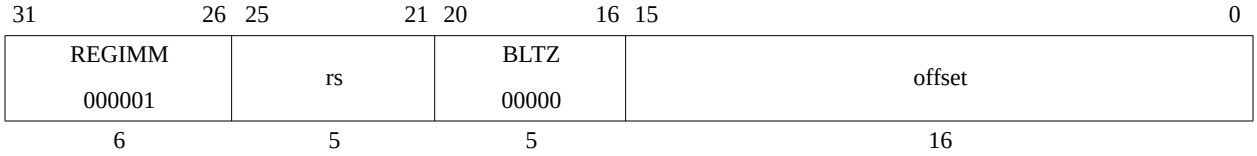
Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

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



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

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if GPR[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 || 0^2)
        condition ← GPR[rs] < 0^GPRLEN
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

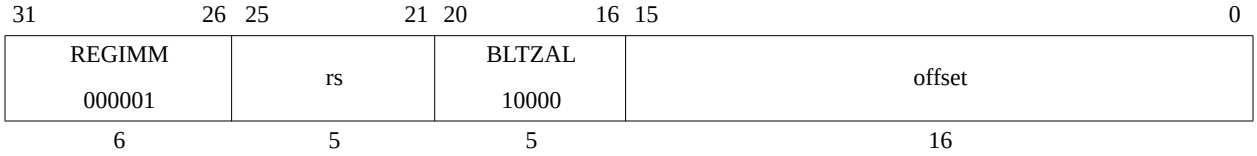
Exceptions:

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.

Branch on Less Than Zero and Link — BLTZAL



Format: `BLTZAL rs, offset` **MIPS32**

Purpose:

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

Description: `if GPR[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:

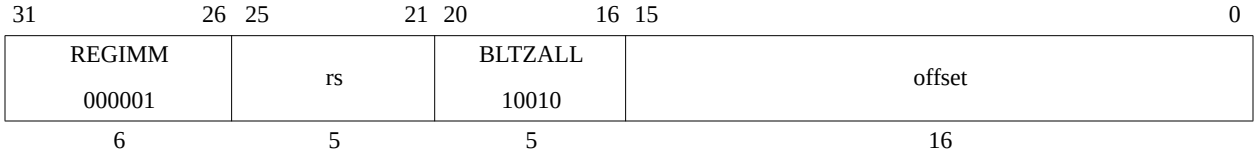
```
I:      target_offset ← sign_extend(offset || 0^2)
        condition ← GPR[rs] < 0^GPRLEN
        GPR[31] ← PC + 8
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

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.



Format: `BLTZALL rs, offset` **MIPS32**

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 GPR[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 || 0^2)
        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.

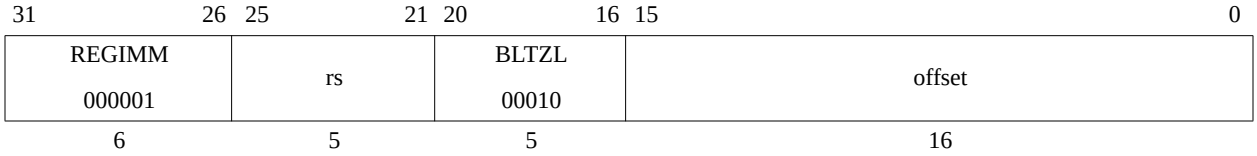
Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

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



Format: `BLTZL rs, offset` **MIPS32**

Purpose:

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

Description: `if GPR[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 || 0^2)
        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.

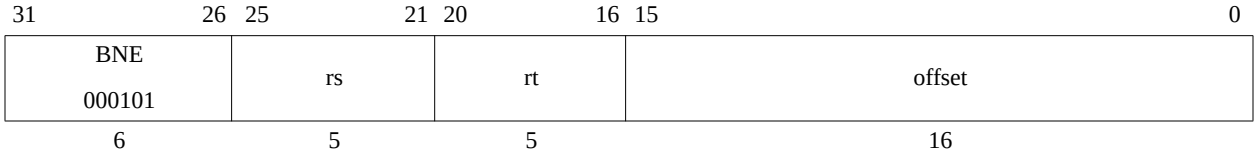
Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability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



Format: `BNE rs, rt, offset` **MIPS32**

Purpose:

To compare GPRs then do a PC-relative conditional branch

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

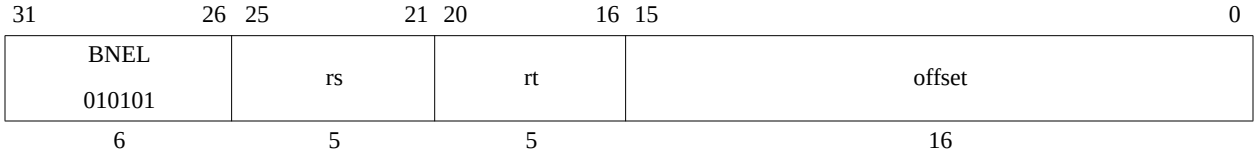
Exceptions:

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.

Branch on Not Equal Likely — BNEL



Format: `BNEL rs, rt, offset` **MIPS32**

Purpose:

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

Description: `if GPR[rs] ≠ GPR[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 || 0^2)
        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.

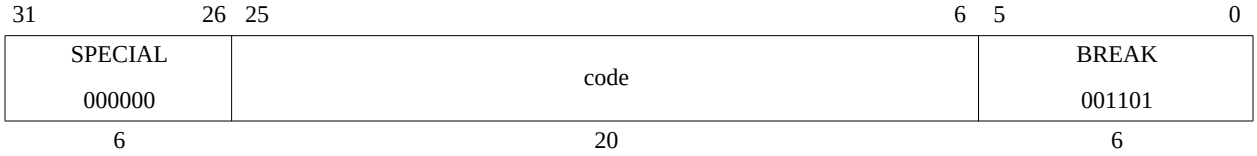
Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

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.

Breakpoint BREAK



Format: `BREAK` **MIPS32**

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:

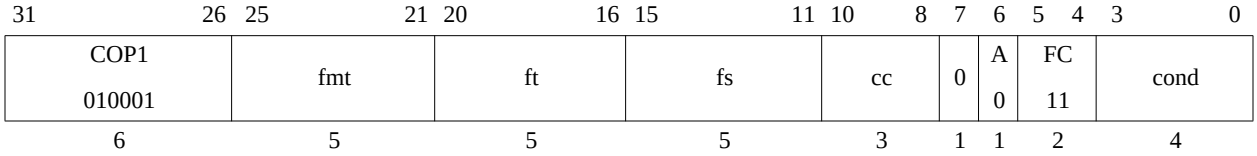
None

Operation:

```
SignalException(Breakpoint)
```

Exceptions:

Breakpoint



Format: `C.cond.S fs, ft (cc = 0 implied)` **MIPS32**

`C.cond.D fs, ft (cc = 0 implied)` **MIPS32**

`C.cond.PS fs, ft(cc = 0 implied)` **MIPS64, MIPS32 Release 2**

`C.cond.S cc, fs, ft` **MIPS32**

`C.cond.D cc, fs, ft` **MIPS32**

`C.cond.PS cc, fs, ft` **MIPS64, MIPS32 Release 2**

Purpose:

To compare FP values and record the Boolean result in a condition code

Description: `FPUConditionCode(cc) ← FPR[fs]` *compare_cond* `FPR[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 3-25 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 3-25 FPU Comparisons Without Special Operand Exceptions

Instruction	Comparison Predicate					Comparison CC Result		Instruction	
		Relation Values						Condition Field	
Cond Mnemonic	Name of Predicate and Logically Negated Predicate (Abbreviation)	>	<	=	?	If Predicate Is True	Inv Op Excp. if QNaN ?	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 3-26 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 **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 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

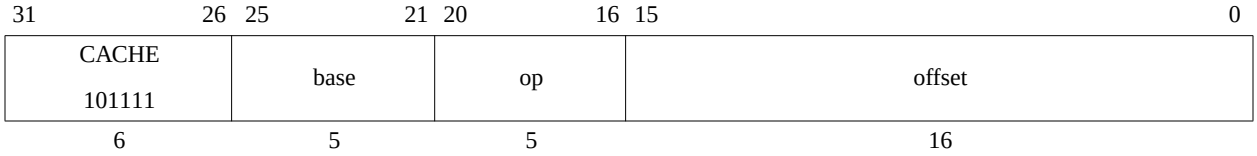
Floating Point Exceptions:

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
```



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

Purpose:

To perform the cache operation specified by op.

Description:

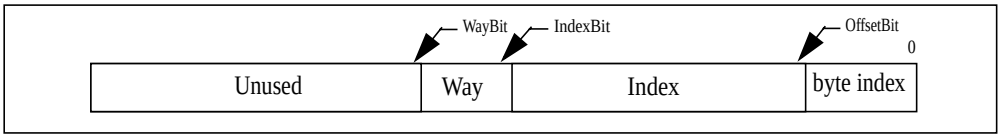
The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

Table 3-27 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. An address translation may or may not be 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. As such, a kseg0 address should always be used for cache operations that require an index. See the Programming Notes section below.
		
		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:
		`OffsetBit ← Log2(BPT)`
		`IndexBit ← Log2(CS / A)`
		`WayBit ← IndexBit + Ceiling(Log2(A))`
		$Way \leftarrow Addr_{WayBit-1..IndexBit}$
		$Index \leftarrow Addr_{IndexBit-1..OffsetBit}$
		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.

The effective address may be an arbitrarily-aligned by address. The CACHE instruction never causes an Address Error Exception due to an non-aligned address.

A Cache Error exception may occur as a by-product 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. However, cache error exceptions must not be triggered by an Index Load Tag or Index Store tag operation, as these operations are used for initialization and diagnostic purposes.

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 3-28 Encoding of Bits[17:16] of CACHE Instruction

Code	Name	Cache
0b00	I	Primary Instruction
0b01	D	Primary Data or Unified Primary
0b10	T	Tertiary
0b11	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 3-29 Encoding of Bits [20:18] of the CACHE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b000	I	Index Invalidate	Index	Set the state of the cache block at the specified index to invalid. 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. For a write-through cache: Set the state of the cache block at the specified index to invalid. 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.	Required
	S, T	Index Writeback Invalidate / Index Invalidate	Index		Optional
0b001	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. This operation must not cause a Cache Error Exception. 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 3-29 Encoding of Bits [20:18] of the CACHE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b010	All	Index Store Tag	Index	Write the tag for the cache block at the specified index from the *TagLo* and *TagHi* Coprocessor 0 registers. This operation must not cause a Cache Error Exception. This required encoding may be used by software to initialize the entire instruction or 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
0b011	All	Implementation Dependent	Unspecified	Available for implementation-dependent operation.	Optional
0b100	I, D	Hit Invalidate	Address	If the cache block contains the specified address, set the state of the cache block to invalid. 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

Table 3-29 Encoding of Bits [20:18] of the CACHE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b101	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. For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid. 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
0b110	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

Table 3-29 Encoding of Bits [20:18] of the CACHE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b111	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

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 operation requires an address, and that address is uncacheable.

The operation of the instruction is **UNPREDICTABLE** if the cache line that contains the CACHE instruction is the target of an invalidate or a writeback invalidate.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

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

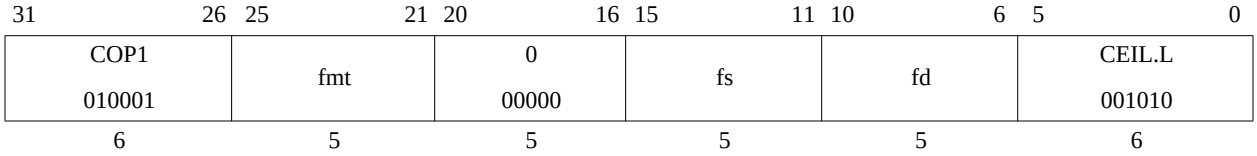
Bus Error Exception

Programming Notes:

For cache operations that require an index, it is implementation dependent whether the effective address or the translated physical address is used as the cache index. Therefore, the index value should always be converted to a kseg0 address by ORing the index with 0x80000000 before being used by the cache instruction. For example, the following code sequence performs a data cache Index Store Tag operation using the index passed in GPR a0:

```
li     a1, 0x80000000          /* Base of kseg0 segment */
or     a0, a0, a1              /* Convert index to kseg0 address */
cache  DCIndexStTag, 0(a1)     /* Perform the index store tag operation */
```

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



Format: `CEIL.L.S fd, fs` **MIPS64, MIPS32 Release 2**

`CEIL.L.D fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

To convert an FP value to 64-bit fixed point, rounding up

Description: `FPR[fd] ← convert_and_round(FPR[fs])`

The value in FPR *fs*, in format *fmt*, is converted to a value in 64-bit long fixed point format and rounding toward +∞ (rounding mode 2). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly, 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))
```

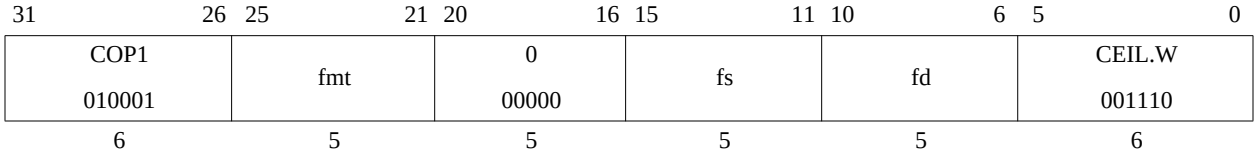
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow

Floating Point Ceiling Convert to Word Fixed Point — CEIL.W.fmt



Format: `CEIL.W.S fd, fs` **MIPS32**
`CEIL.W.D fd, fs` **MIPS32**

Purpose:

To convert an FP value to 32-bit fixed point, rounding up

Description: `FPR[fd] ← convert_and_round(FPR[fs])`

The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format and rounding toward +∞ (rounding mode 2). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly, 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))
```

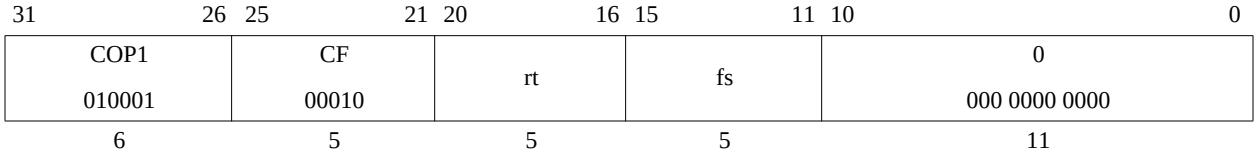
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow

Move Control Word From Floating Point — CFC1



Format: `CFC1 rt, fs` **MIPS32**

Purpose:

To copy a word from an FPU control register to a GPR

Description: `GPR[rt] ← FP_Control[FPR[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 ← 0^24 || FCSR_31..25 || FCSR_23
elseif fs = 26 then
   temp ← 0^14 || FCSR_17..12 || 0^5 || FCSR_6..2 || 0^2
elseif fs = 28 then
   temp ← 0^20 || FCSR_11.7 || 0^4 || FCSR_24 || FCSR_1..0
elseif fs = 31 then
   temp ← FCSR
else
   temp ← UNPREDICTABLE
endif
GPR[rt] ← sign_extend(temp)
```

Exceptions:

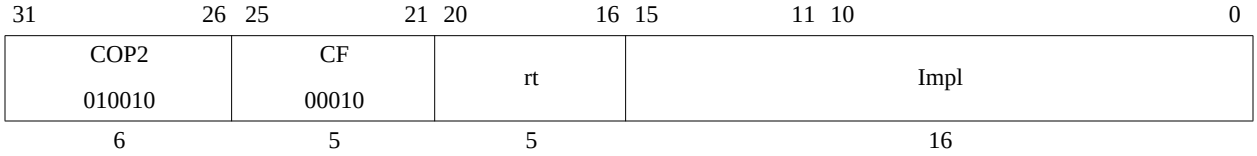
Coprocessor Unusable, Reserved Instruction

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.

Move Control Word From Coprocessor 2 — CFC2



Format: `CFC2 rt, rd` **MIPS32**

The syntax shown above is an example using CFC1 as a model. The specific syntax is implementation dependent.

Purpose:

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

Description: `GPR[rt] ← CP2CCR[Impl]`

Copy the 32-bit word from the Coprocessor 2 control register denoted by the *Impl* field, sign-extending it to 64 bits. The interpretation of the *Impl* field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:

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

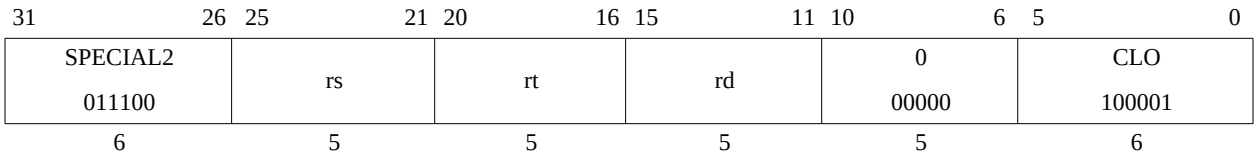
Operation:

```
temp ← CP2CCR[Impl]
GPR[rt] ← sign_extend(temp)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Count Leading Ones in Word — CLO



Format: `CLO rd, rs` **MIPS32**

Purpose:

To Count the number of leading ones in a word

Description: `GPR[rd] ← count_leading_ones GPR[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**.

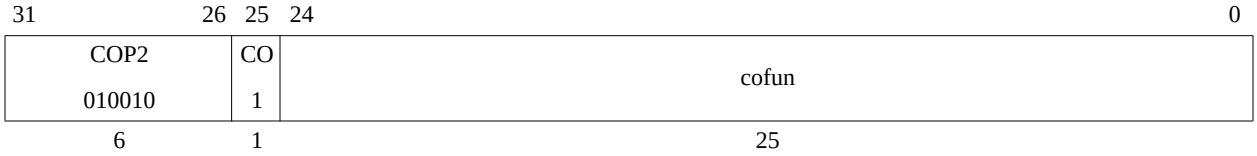
Operation:

```
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

Coprocessor Operation to Coprocessor 2 — COP2



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:

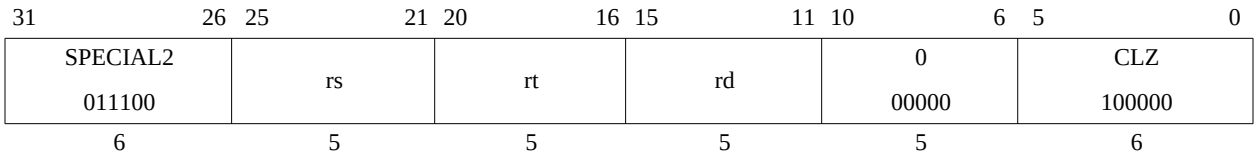
Operation:

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

Exceptions:

Coprocessor Unusable
Reserved Instruction

Count Leading Zeros in Word — CLZ



Format: `CLZ rd, rs` **MIPS32**

Purpose

Count the number of leading zeros in a word

Description: `GPR[rd] ← count_leading_zeros GPR[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**.

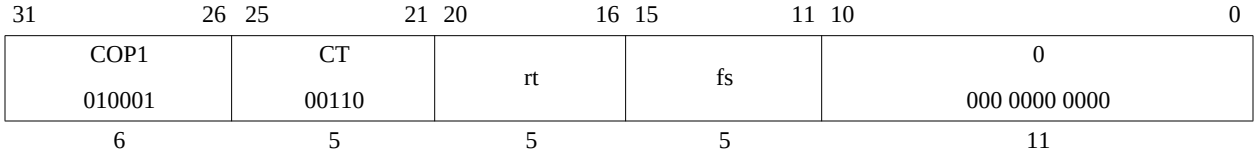
Operation:

```
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

Move Control Word to Floating Point — CTC1



Format: `CTC1 rt, fs` **MIPS32**

Purpose:

To copy a word from a GPR to an FPU control register

Description: `FP_Control[fs] ← GPR[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 and the *EPC* register contains the address of the CTC1 instruction.

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 /* FCCR */
   if temp31..8 ≠ 0^24 then
      UNPREDICTABLE
   else
      FCSR ← temp7..1 || FCSR24 || temp0 || FCSR22..0
   endif
elseif fs = 26 then /* FEXR */
   if temp22..18 ≠ 0 then
      UNPREDICTABLE
   else
      FCSR ← FCSR31..18 || temp17..12 || FCSR11..7 ||
      temp6..2 || FCSR1..0
   endif
elseif fs = 28 then /* FENR */
   if temp22..18 ≠ 0 then
      UNPREDICTABLE
   else
      FCSR ← FCSR31..25 || temp2 || FCSR23..12 || temp11..7
      || FCSR6..2 || temp1..0
   endif
elseif fs = 31 then /* FCSR */
   if temp22..18 ≠ 0 then
      UNPREDICTABLE
   else
      FCSR ← temp
   endif
else
   UNPREDICTABLE
endif
```

`CheckFPException()`**Exceptions:**

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

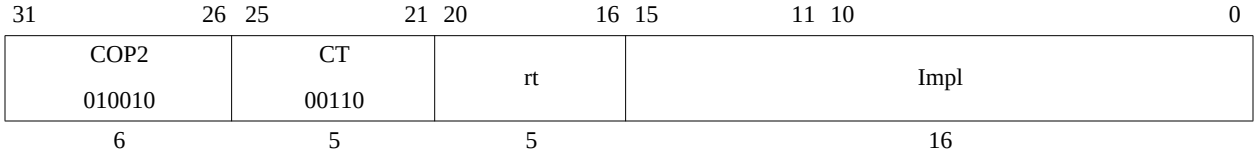
Unimplemented Operation, Invalid Operation, Division-by-zero, Inexact, Overflow, Underflow

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



Format: `CTC2 rt, rd` **MIPS32**

The syntax shown above is an example using CTC1 as a model. The specific syntax is implementation dependent.

Purpose:

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

Description: `CP2CCR[Impl] ← GPR[rt]`

Copy the low word from GPR *rt* into the Coprocessor 2 control register denoted by the *Impl* field. The interpretation of the *Impl* field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:

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

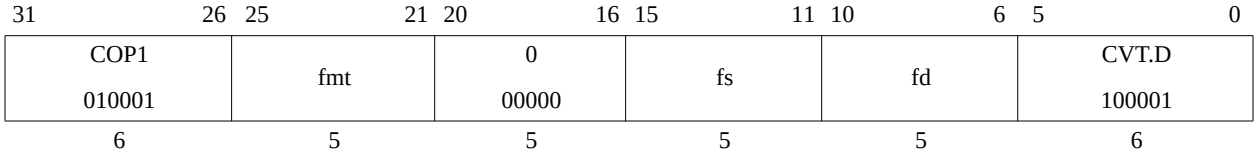
Operation:

```
temp ← GPR[rt]31..0
CP2CCR[Impl] ← temp
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Convert to Double Floating Point — CVT.D.fmt



Format: `CVT.D.S fd, fs` **MIPS32**

`CVT.D.W fd, fs` **MIPS32**

`CVT.D.L fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

To convert an FP or fixed point value to double FP

Description: `FPR[fd] ← convert_and_round(FPR[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**.

For CVT.D.L, the result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

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

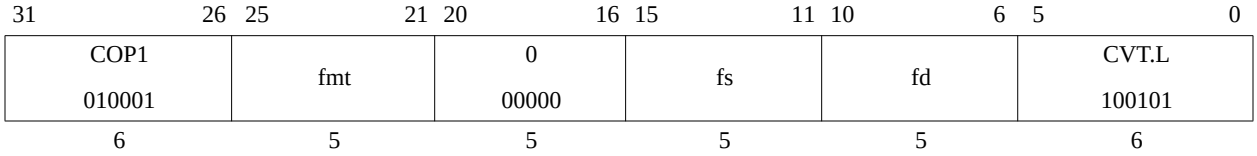
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

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



Format: `CVT.L.S fd, fs` **MIPS64, MIPS32 Release 2**

`CVT.L.D fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

To convert an FP value to a 64-bit fixed point

Description: `FPR[fd] ← convert_and_round(FPR[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**.

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

Operation:

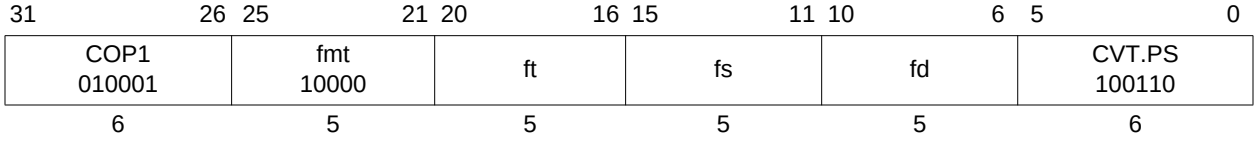
```
StoreFPR (fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow



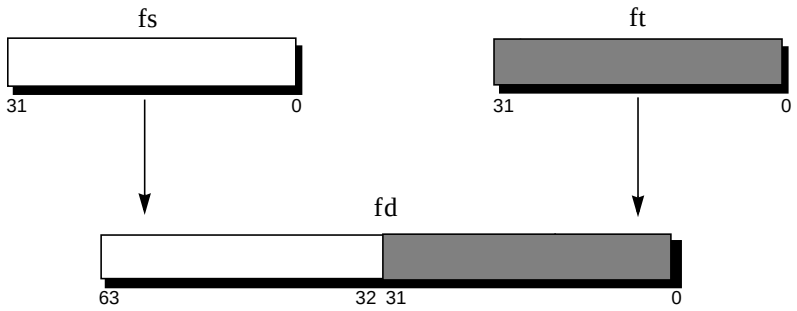
Format: `CVT.PS.S fd, fs, ft` **MIPS64, MIPS32 Release 2**

Purpose:

To convert two FP values to a paired single value

Description: $FPR[fd] \leftarrow FPR[fs]_{31..0} \mid\mid FPR[ft]_{31..0}$

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))
```

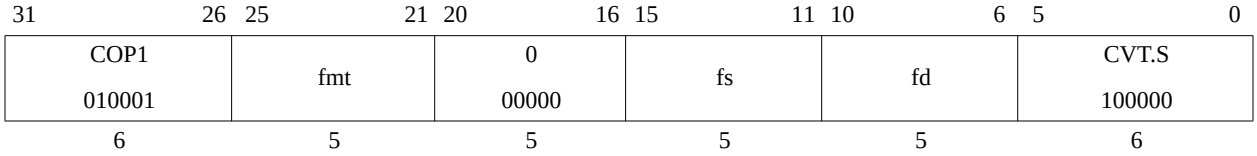
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation

Floating Point Convert to Single Floating Point — CVT.S.fmt



Format: `CVT.S.D fd, fs` **MIPS32**

`CVT.S.W fd, fs` **MIPS32**

`CVT.S.L fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

To convert an FP or fixed point value to single FP

Description: `FPR[fd] ← convert_and_round(GPR[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**.

For CVT.S.L, the result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

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

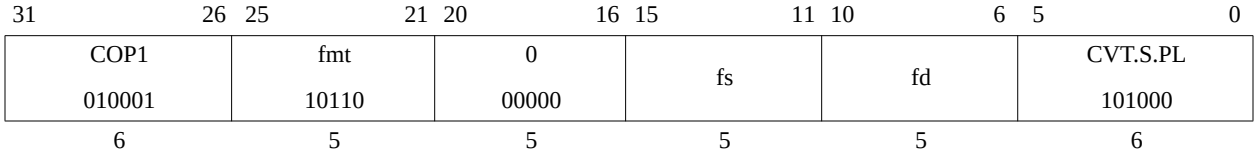
Exceptions:

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, MIPS32 Release 2**

Purpose:

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

Description: `GPR[fd] ← convert_and_round(GPR[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:

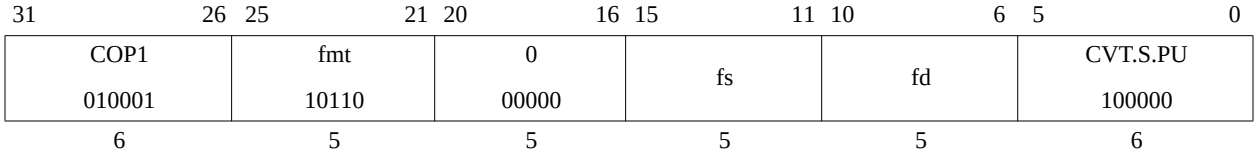
```
StoreFPR (fd, S, ConvertFmt(ValueFPR(fs, PS), PL, S))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow, Underflow



Format: `CVT.S.PU fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

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

Description: `FPR[fd] ← convert_and_round(FPR[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))
```

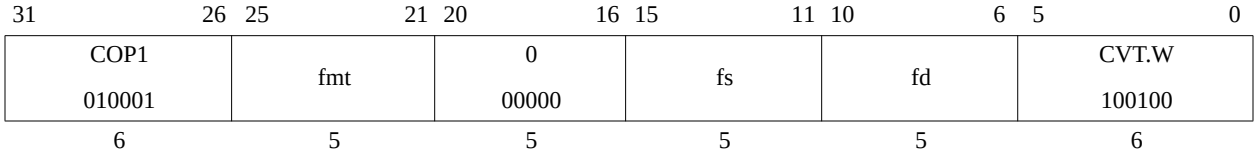
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow, Underflow

Floating Point Convert to Word Fixed Point — CVT.W.fmt



Format: `CVT.W.S fd, fs` **MIPS32**
`CVT.W.D fd, fs` **MIPS32**

Purpose:

To convert an FP value to 32-bit fixed point

Description: `FPR[fd] ← convert_and_round(FPR[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))
```

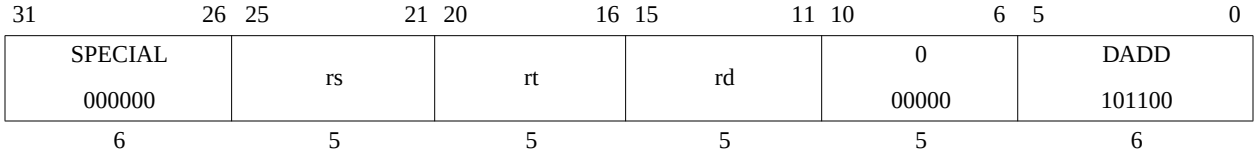
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow

Doubleword Add DADD



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

Purpose:

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

Description: `GPR[rd] ← GPR[rs] + GPR[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 (temp64 ≠ temp63) then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp63..0
endif
```

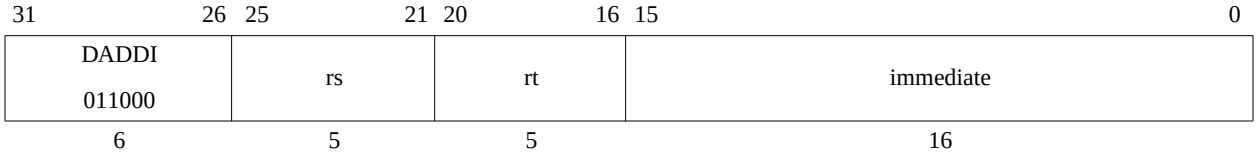
Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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

Doubleword Add Immediate DADDI



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

Purpose:

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

Description: `GPR[rt] ← GPR[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 (temp64 ≠ temp63) then
   SignalException(IntegerOverflow)
else
   GPR[rt] ← temp63..0
endif
```

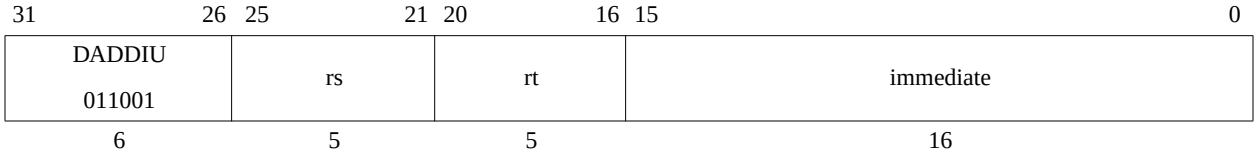
Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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

Doubleword Add Immediate Unsigned DADDIU



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

Purpose:

To add a constant to a 64-bit integer

Description: `GPR[rt] ← GPR[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)
```

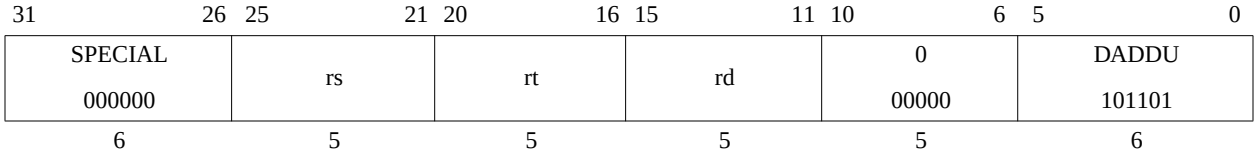
Exceptions:

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 DADDU



Format: `DADDU rd, rs, rt` **MIPS64**

Purpose:

To add 64-bit integers

Description: `GPR[rd] ← GPR[rs] + GPR[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]
```

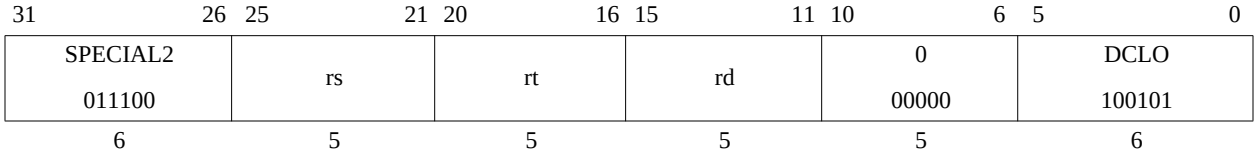
Exceptions:

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.

Count Leading Ones in Doubleword — DCLO



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

Purpose:

To count the number of leading ones in a doubleword

Description: `GPR[rd] ← count_leading_ones GPR[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:

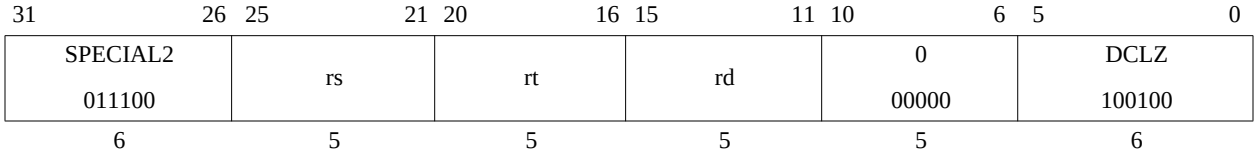
To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the *rt* and *rd* fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values.

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



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

Purpose:

To count the number of leading zeros in a doubleword

Description: `GPR[rd] ← count_leading_zeros GPR[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.

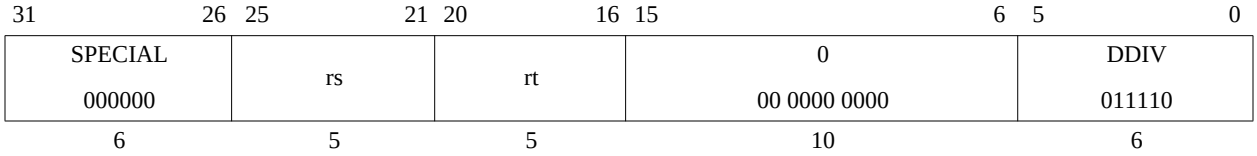
Operation:

```
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



Format: `DDIV rs, rt` **MIPS64**

Purpose:

To divide 64-bit signed integers

Description: `(LO, HI) ← GPR[rs] / GPR[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

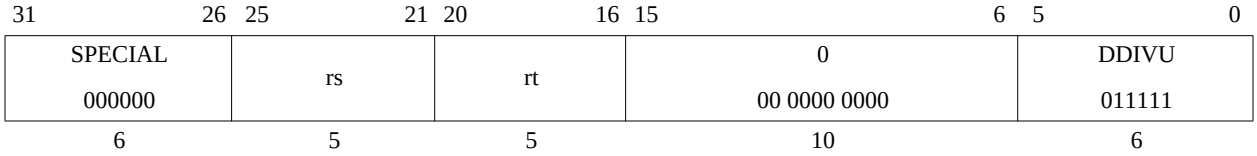
Programming Notes:

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.

Doubleword Divide Unsigned — DDIVU



Format: `DDIVU rs, rt` **MIPS64**

Purpose:

To divide 64-bit unsigned integers

Description: `(LO, HI) ← GPR[rs] / GPR[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 ← q63..0
HI ← r63..0
```

Exceptions:

Reserved Instruction

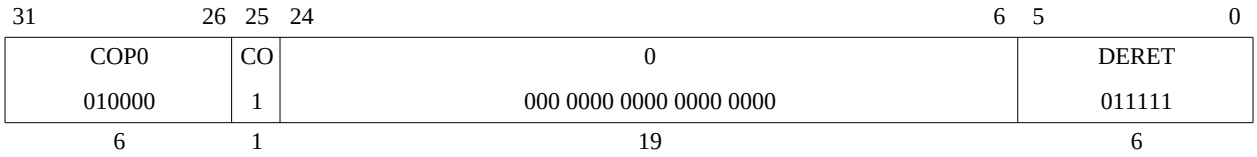
Programming Notes:

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 clears execution and instruction hazards, 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 exists that must be removed via software insertion of the appropriate number of SSNOP instructions (for implementations of Release 1 of the Architecture) or by an EHB, or other execution hazard clearing instruction (for implementations of Release 2 of the Architecture).

DERET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the DERET returns.

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.

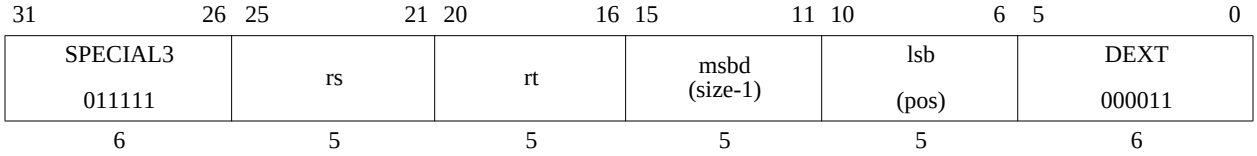
Operation:

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

Exceptions:

Coprocessor Unusable Exception
Reserved Instruction Exception

Doubleword Extract Bit Field — DEXT



Format: `dext rt, rs, pos, size` **MIPS64 Release 2**

Purpose:

To extract a bit field from GPR *rs* and store it right-justified into GPR *rt*.

Description: `GPR[rt] ← ExtractField(GPR[rs], msbd, lsb)`

The bit field starting at bit *pos* and extending for *size* bits is extracted from GPR *rs* and stored zero-extended and right-justified in GPR *rt*. The assembly language arguments pos and size are converted by the assembler to the instruction fields *msbd* (the most significant bit of the destination field in GPR rt), in instruction bits 15..11, and *lsb* (least significant bit of the source field in GPR rs), in instruction bits 10..6, as follows:

```
msbd ← size-1
lsb ← pos
msb ← lsb+msbd
```

For this instruction, the values of *pos* and *size* must satisfy all of the following relations:

```
0 ≤ pos < 32
0 < size ≤ 32
0 < pos+size ≤ 63
```

Figure 3-3 shows the symbolic operation of the instruction.

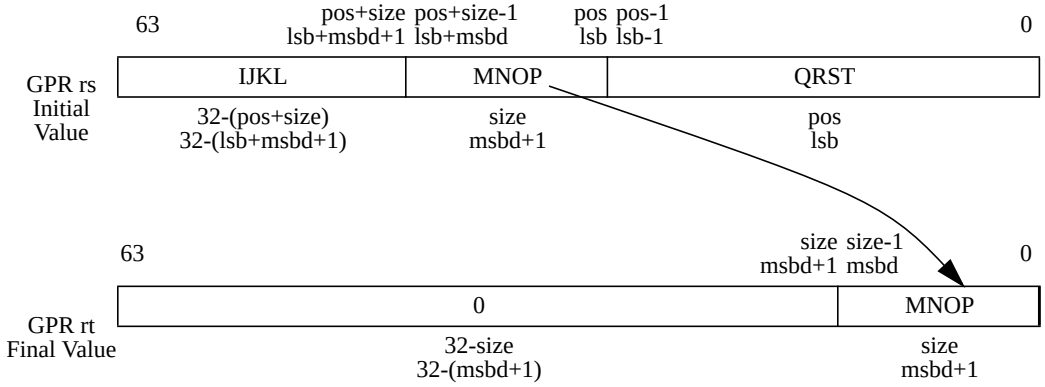


Figure 3-3 Operation of the DEXT Instruction

Three instructions are required to access any legal bit field within the doubleword, as a function of the *msb* (as derived from *msbd* and *lsb*) and *lsb* of the field (which implies restrictions on *pos* and *size*), as follows:

msbd	*lsb*	*msb*	*pos*	*size*	Instruction	Comment
$0 \le msbd < 32$	$0 \le lsb < 32$	$0 \le msb < 63$	$0 \le pos < 32$	$1 \le size \le 32$	DEXT	The field is 32 bits or less and starts in the right-most word of the doubleword
$0 \le msbd < 32$	$32 \le lsb < 64$	$32 \le msb < 64$	$32 \le pos < 64$	$1 \le size \le 32$	DEXTU	The field is 32 bits or less and starts in the left-most word of the doubleword
$32 \le msbd < 64$	$0 \le lsb < 32$	$32 \le msb < 64$	$0 \le pos < 32$	$32 < size \le 64$	DEXTM	The field is larger than 32 bits and starts in the right-most word of the doubleword

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Because of the limits on the values of *msbd* and *lsb*, there is no **UNPREDICTABLE** case for this instruction.

Operation:

```
GPR[rt] ← 0^(63-(msbd+1)) || GPR[rs]_(msbd+lsb..lsb)
```

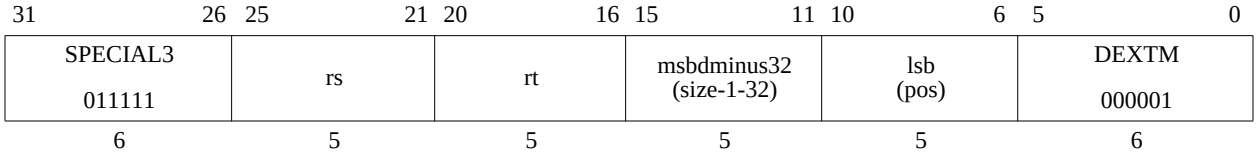
Exceptions:

Reserved Instruction

Programming Notes

The assembler will accept any value of *pos* and *size* that satisfies the relationship 0 < pos+size ≤ 64 and emit DEXT, DEXTM, or DEXTU as appropriate to the values. Programmers should always specify the DEXT mnemonic and let the assembler select the instruction to use.

Doubleword Extract Bit Field Middle — DEXTM



Format: `dextm rt, rs, pos, size` **MIPS64 Release 2**

Purpose:

To extract a bit field from GPR *rs* and store it right-justified into GPR *rt*.

Description: `GPR[rt] ← ExtractField(GPR[rs], msbd, lsb)`

The bit field starting at bit *pos* and extending for *size* bits is extracted from GPR *rs* and stored zero-extended and right-justified in GPR *rt*. The assembly language arguments pos and size are converted by the assembler to the instruction fields *msbdminus32* (the most significant bit of the destination field in GPR rt, minus 32), in instruction bits 15..11, and *lsb* (least significant bit of the source field in GPR rs), in instruction bits 10..6, as follows:

```
msbdminus32 ← size-1-32
lsb ← pos
msbd ← msbdminus32 + 32
msb ← lsb+msbd
```

For this instruction, the values of *pos* and *size* must satisfy all of the following relations:

```
0 ≤ pos < 32
32 < size ≤ 64
32 < pos+size ≤ 64
```

Figure 3-4 shows the symbolic operation of the instruction.

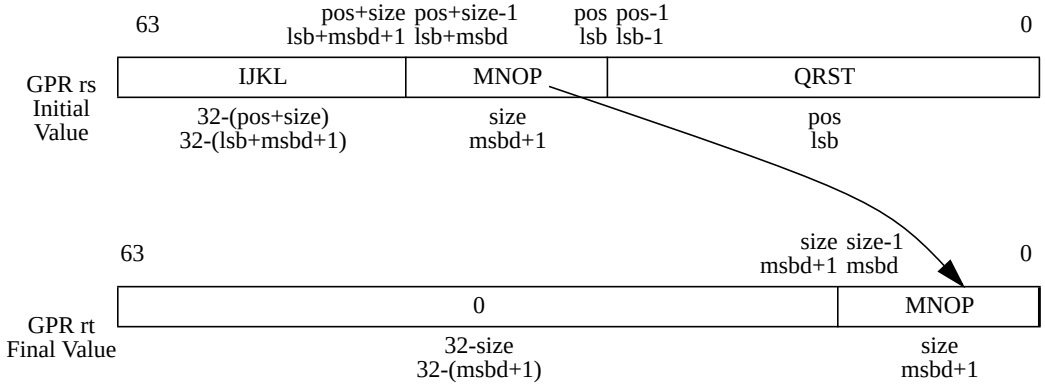


Figure 3-4 Operation of the DEXTM Instruction

Three instructions are required to access any legal bit field within the doubleword, as a function of the *msb* (as derived from *msbd* and *lsb*) and *lsb* of the field (which implies restrictions on *pos* and *size*), as follows:

msbd	*lsb*	*msb*	*pos*	*size*	Instruction	Comment
$0 \le msbd < 32$	$0 \le lsb < 32$	$0 \le msb < 63$	$0 \le pos < 32$	$1 \le size \le 32$	DEXT	The field is 32 bits or less and starts in the right-most word of the doubleword
$0 \le msbd < 32$	$32 \le lsb < 64$	$32 \le msb < 64$	$32 \le pos < 64$	$1 \le size \le 32$	DEXTU	The field is 32 bits or less and starts in the left-most word of the doubleword
$32 \le msbd < 64$	$0 \le lsb < 32$	$32 \le msb < 64$	$0 \le pos < 32$	$32 < size \le 64$	DEXTM	The field is larger than 32 bits and starts in the right-most word of the doubleword

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

The operation is **UNPREDICTABLE** if (*lsb* + *msbd* + 1) > 64.

Operation:

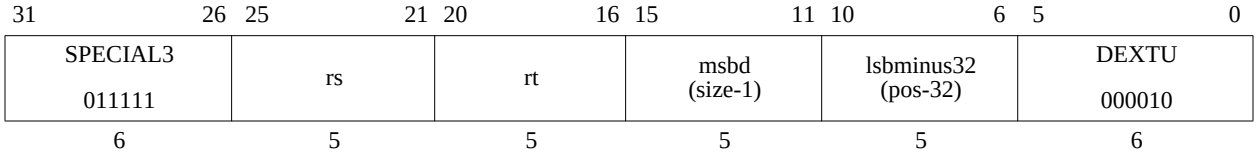
```
msbd ← msbdminus32 + 32
if ((lsb + msbd + 1) > 64) then
   UNPREDICTABLE
endif
GPR[rt] ← 0^(63-(msbd+1)) || GPR[rs]_(msbd+lsb..pos)
```

Exceptions:

Reserved Instruction

Programming Notes

The assembler will accept any value of *pos* and *size* that satisfies the relationship 0 < pos+size ≤ 64 and emit DEXT, DEXTM, or DEXTU as appropriate to the values. Programmers should always specify the DEXT mnemonic and let the assembler select the instruction to use.



Format: `dextu rt, rs, pos, size` **MIPS64 Release 2**

Purpose:

To extract a bit field from GPR *rs* and store it right-justified into GPR *rt*.

Description: `GPR[rt] ← ExtractField(GPR[rs], msbd, lsb)`

The bit field starting at bit *pos* and extending for *size* bits is extracted from GPR *rs* and stored zero-extended and right-justified in GPR *rt*. The assembly language arguments pos and size are converted by the assembler to the instruction fields *msbd* (the most significant bit of the destination field in GPR rt), in instruction bits 15..11, and *lsbminus32* (least significant bit of the source field in GPR rs, minus32), in instruction bits 10..6, as follows:

```
msbd ← size-1
lsbminus32 ← pos-32
lsb ← lsbminus32 + 32
msb ← lsb+msbd
```

For this instruction, the values of *pos* and *size* must satisfy all of the following relations:

```
32 ≤ pos < 64
0 < size ≤ 32
32 < pos+size ≤ 64
```

Figure 3-5 shows the symbolic operation of the instruction.

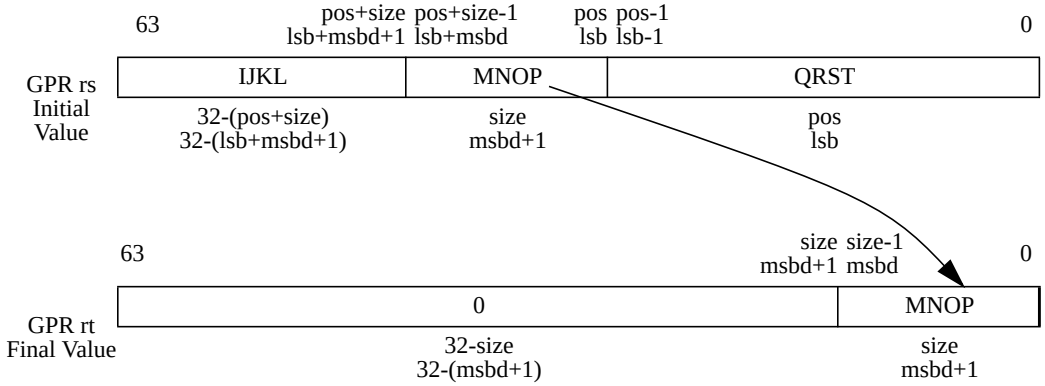


Figure 3-5 Operation of the DEXTU Instruction

Three instructions are required to access any legal bit field within the doubleword, as a function of the *msb* (as derived from *msbd* and *lsb*) and *lsb* of the field (which implies restrictions on *pos* and *size*), as follows:

msbd	*lsb*	*msb*	*pos*	*size*	Instruction	Comment
$0 \le msbd < 32$	$0 \le lsb < 32$	$0 \le msb < 63$	$0 \le pos < 32$	$1 \le size \le 32$	DEXT	The field is 32 bits or less and starts in the right-most word of the doubleword
$0 \le msbd < 32$	$32 \le lsb < 64$	$32 \le msb < 64$	$32 \le pos < 64$	$1 \le size \le 32$	DEXTU	The field is 32 bits or less and starts in the left-most word of the doubleword
$32 \le msbd < 64$	$0 \le lsb < 32$	$32 \le msb < 64$	$0 \le pos < 32$	$32 < size \le 64$	DEXTM	The field is larger than 32 bits and starts in the right-most word of the doubleword

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

The operation is **UNPREDICTABLE** if (*lsb* + *msbd* + 1) > 64.

Operation:

```
lsb ← lsbminus32 + 32
if ((lsb + msbd + 1) > 64) then
   UNPREDICTABLE
endif
GPR[rt] ← 0^(63-(msbd+1)) || GPR[rs]_(msbd+lsb..pos)
```

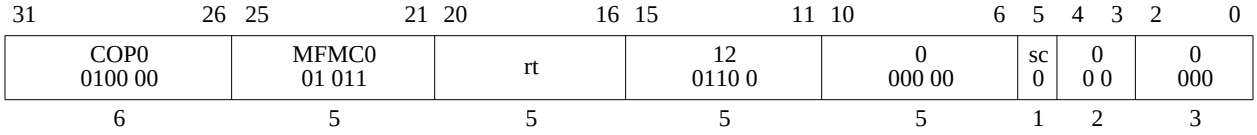
Exceptions:

Reserved Instruction

Programming Notes

The assembler will accept any value of *pos* and *size* that satisfies the relationship 0 < pos+size ≤ 64 and emit DEXT, DEXTM, or DEXTU as appropriate to the values. Programmers should always specify the DEXT mnemonic and let the assembler select the instruction to use.

Disable Interrupts DI



Format: `DI` **MIPS32 Release 2**

`DI rt` **MIPS32 Release 2**

Purpose:

To return the previous value of the *Status* register and disable interrupts. If DI is specified without an argument, GPR r0 is implied, which discards the previous value of the Status register.

Description: `GPR[rt] ← Status; Status`$_{IE}$ `← 0`

The current value of the *Status* register is sign-extended and loaded into general register *rt*. The Interrupt Enable (IE) bit in the *Status* register is then cleared.

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

This operation specification is for the general interrupt enable/disable operation, with the *sc* field as a variable. The individual instructions DI and EI have a specific value for the *sc* field.

```
data ← Status
GPR[rt] ← sign_extend(data)
Status_IE ← 0
```

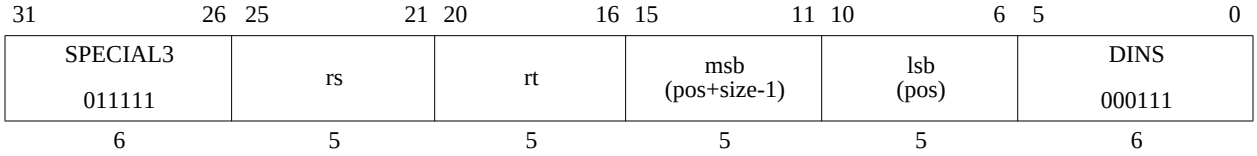
Exceptions:

Coprocessor Unusable
Reserved Instruction (Release 1 implementations)

Programming Notes:

The effects of this instruction are identical to those accomplished by the sequence of reading *Status* into a GPR, clearing the IE bit, and writing the result back to *Status*. Unlike the multiple instruction sequence, however, the DI instruction can not be aborted in the middle by an interrupt or exception.

This instruction creates an execution hazard between the change to the Status register and the point where the change to the interrupt enable takes effect. This hazard is cleared by the EHB, JALR.HB, JR.HB, or ERET instructions. Software must not assume that a fixed latency will clear the execution hazard.



Format: `dins rt, rs, pos, size` **MIPS64 Release 2**

Purpose:

To merge a right-justified bit field from GPR *rs* into a specified position in GPR *rt*.

Description: `GPR[rt] ← InsertField(GPR[rt], GPR[rs], msb, lsb)`

The right-most *size* bits from GPR *rs* are merged into the value from GPR *rt* starting at bit position *pos*. The result is placed back in GPR *rt*. The assembly language arguments *pos* and *size* are converted by the assembler to the instruction fields *msb* (the most significant bit of the field), in instruction bits 15..11, and *lsb* (least significant bit of the field), in instruction bits 10..6, as follows:

```
msb ← pos+size-1
lsb ← pos
```

For this instruction, the values of *pos* and *size* must satisfy all of the following relations:

```
0 ≤ pos < 32
0 < size ≤ 32
0 < pos+size ≤ 32
```

Figure 3-6 shows the symbolic operation of the instruction.

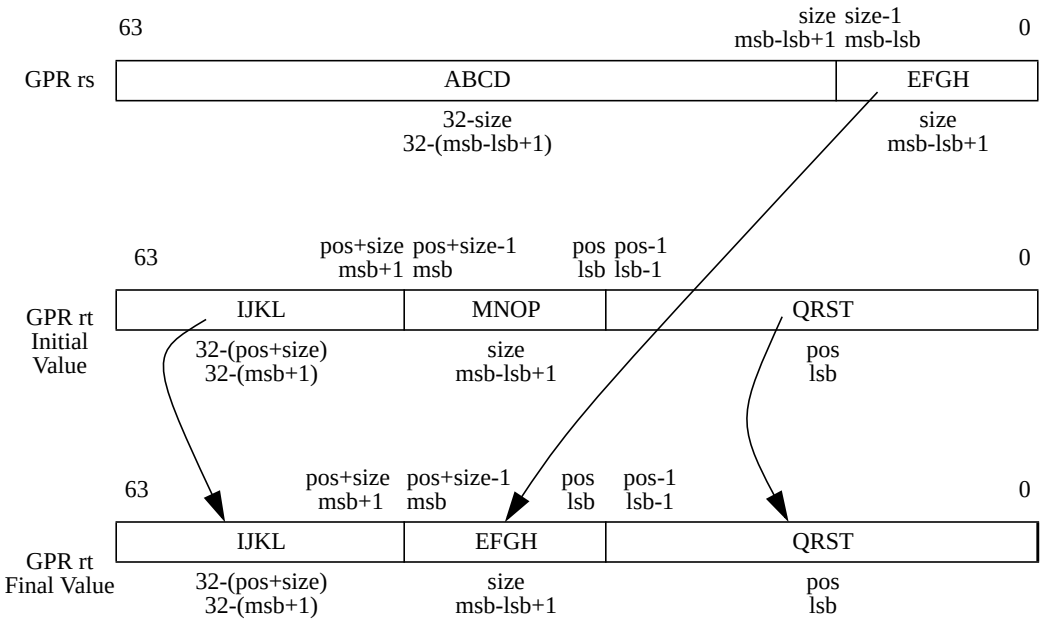


Figure 3-6 Operation of the DINS Instruction

Three instructions are required to access any legal bit field within the doubleword, as a function of the *msb* and *lsb* of the field (which implies restrictions on *pos* and *size*), as follows:

msb	*lsb*	*pos*	*size*	Instruction	Comment
$0 \le msb < 32$	$0 \le lsb < 32$	$0 \le pos < 32$	$1 \le size \le 32$	DINS	The field is entirely contained in the right-most word of the doubleword
$32 \le msb < 64$	$0 \le lsb < 32$	$0 \le pos < 32$	$2 \le size \le 64$	DINSM	The field straddles the words of the doubleword
$32 \le msb < 64$	$32 \le lsb < 64$	$32 \le pos < 64$	$1 \le size \le 32$	DINSU	The field is entirely contained in the left-most word of the doubleword

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

The operation is **UNPREDICTABLE** if *lsb* > *msb.*

Operation:

```
if (lsb > msb) then
    UNPREDICTABLE
endif
GPR[rt] ← GPR[rt]63..msb+1 || GPR[rs]msb-lsb..0 || GPR[rt]lsb-1..0
```

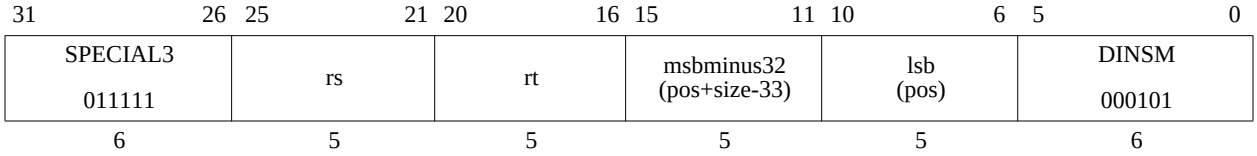
Exceptions:

Reserved Instruction

Programming Notes

The assembler will accept any value of *pos* and *size* that satisfies the relationship $0 < pos+size \le 64$ and emit DINS, DINSM, or DINSU as appropriate to the values. Programmers should always specify the DINS mnemonic and let the assembler select the instruction to use.

Doubleword Insert Bit Field Middle — DINSM



Format: `dinsm rt, rs, pos, size` **MIPS64 Release 2**

Purpose:

To merge a right-justified bit field from GPR *rs* into a specified position in GPR *rt*.

Description: `GPR[rt] ← InsertField(GPR[rt], GPR[rs], msb, lsb)`

The right-most *size* bits from GPR *rs* are inserted into the value from GPR *rt* starting at bit position *pos*. The result is placed back in GPR *rt*. The assembly language arguments *pos* and *size* are converted by the assembler to the instruction fields *msbminus32* (the most significant bit of the field, minus 32), in instruction bits 15..11, and *lsb* (least significant bit of the field), in instruction bits 10..6, as follows:

```
msbminus32 ← pos+size-1-32
lsb ← pos
msb ← msbminus32 + 32
```

For this instruction, the values of *pos* and *size* must satisfy all of the following relations:

```
0 ≤ pos < 32
2 ≤ size ≤ 64
32 < pos+size ≤ 64
```

Figure 3-7 shows the symbolic operation of the instruction.

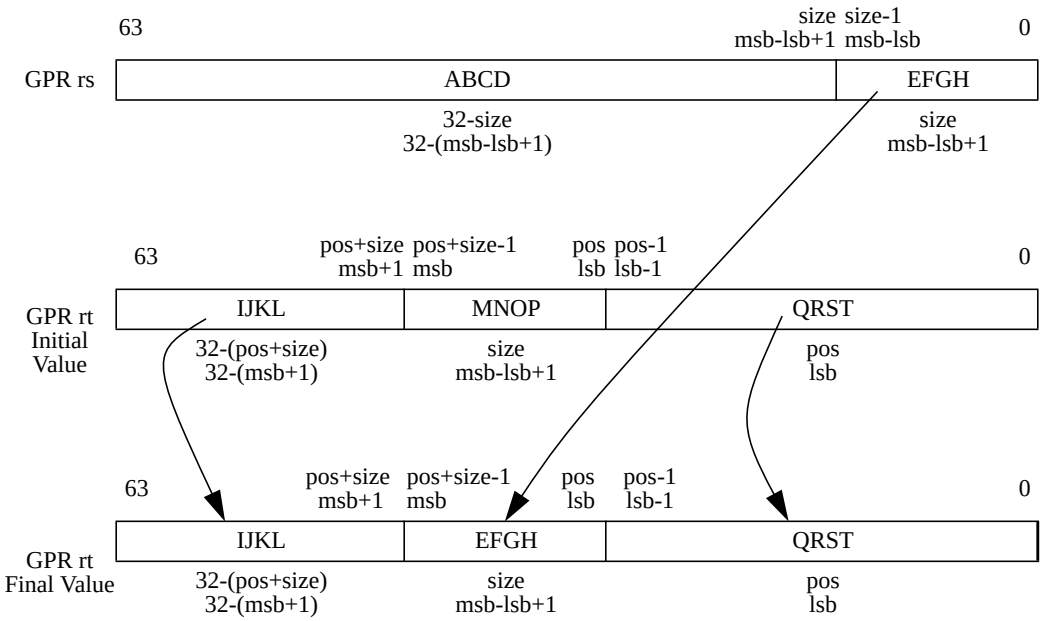


Figure 3-7 Operation of the DINSM Instruction

Three instructions are required to access any legal bit field within the doubleword, as a function of the *msb* and *lsb* of the field (which implies restrictions on *pos* and *size*), as follows:

msb	*lsb*	*pos*	*size*	Instruction	Comment
$0 \le msb < 32$	$0 \le lsb < 32$	$0 \le pos < 32$	$1 \le size \le 32$	DINS	The field is entirely contained in the right-most word of the doubleword
$32 \le msb < 64$	$0 \le lsb < 32$	$0 \le pos < 32$	$2 \le size \le 64$	DINSM	The field straddles the words of the doubleword
$32 \le msb < 64$	$32 \le lsb < 64$	$32 \le pos < 64$	$1 \le size \le 32$	DINSU	The field is entirely contained in the left-most word of the doubleword

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Because of the instruction format, *lsb* can never be greater than *msb*, so there is no **UNPREDICATABLE** case for this instruction.

Operation:

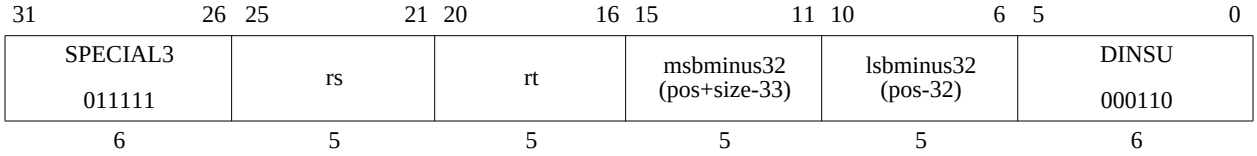
```
msb ← msbminus32 + 32
GPR[rt] ← GPR[rt]63..msb+1 || GPR[rs]msb-lsb..0 || GPR[rt]lsb-1..0
```

Exceptions:

Reserved Instruction

Programming Notes

The assembler will accept any value of *pos* and *size* that satisfies the relationship $0 < pos+size \le 64$ and emit DINS, DINSM, or DINSU as appropriate to the values. Programmers should always specify the DINS mnemonic and let the assembler select the instruction to use.



Format: `dinsu rt, rs, pos, size` **MIPS64 Release 2**

Purpose:

To merge a right-justified bit field from GPR *rs* into a specified position in GPR *rt*.

Description: `GPR[rt] ← InsertField(GPR[rt], GPR[rs], msb, lsb)`

The right-most *size* bits from GPR *rs* are inserted into the value from GPR *rt* starting at bit position *pos*. The result is placed back in GPR *rt*. The assembly language arguments *pos* and *size* are converted by the assembler to the instruction fields *msbminus32* (the most significant bit of the field, minus 32), in instruction bits 15..11, and *lsbminus32* (least significant bit of the field, minus 32), in instruction bits 10..6, as follows:

```
msbminus32 ← pos+size-1-32
lsbminus32 ← pos-32
msb ← msbminus32 + 32
lsb ← lsbminus32 + 32
```

For this instruction, the values of *pos* and *size* must satisfy all of the following relations:

```
32 ≤ pos < 64
1 ≤ size ≤ 32
32 < pos+size ≤ 64
```

Figure 3-8 shows the symbolic operation of the instruction.

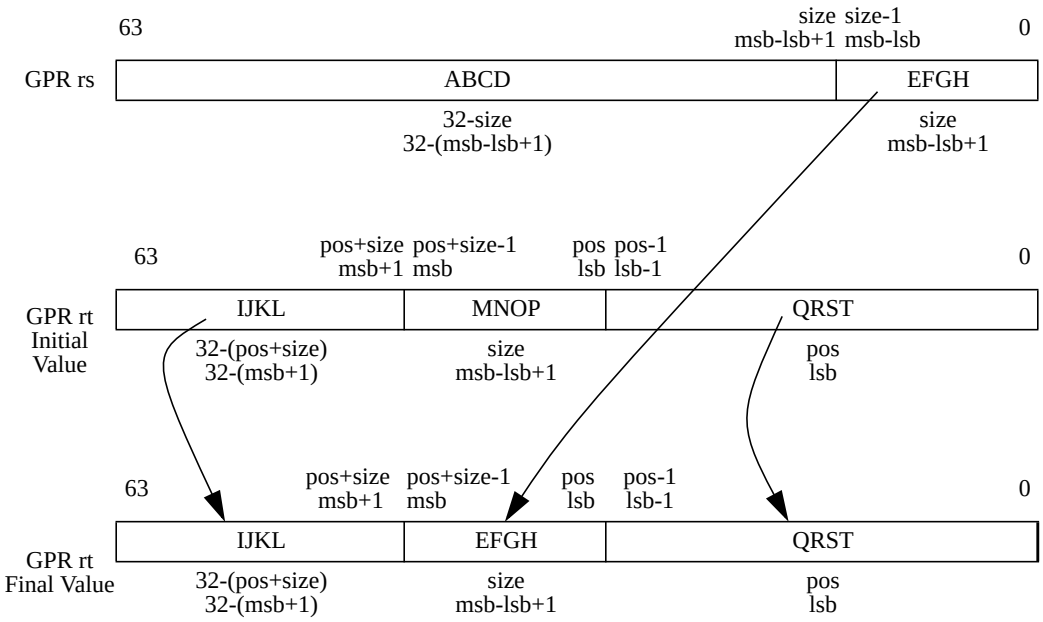


Figure 3-8 Operation of the DINSU Instruction

Three instructions are required to access any legal bit field within the doubleword, as a function of the *msb* and *lsb* of the field (which implies restrictions on *pos* and *size*), as follows:

msb	*lsb*	*pos*	*size*	Instruction	Comment
0 ≤ *msb* < 32	0 ≤ *lsb* < 32	0 ≤ pos < 32	1 ≤ size ≤ 32	DINS	The field is entirely contained in the right-most word of the doubleword
32 ≤ *msb* < 64	0 ≤ *lsb* < 32	0 ≤ pos < 32	2 ≤ size ≤ 64	DINSM	The field straddles the words of the doubleword
32 ≤ *msb* < 64	32 ≤ *lsb* < 64	32 ≤ pos < 64	1 ≤ size ≤ 32	DINSU	The field is entirely contained in the left-most word of the doubleword

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

The operation is **UNPREDICTABLE** if *lsb* > *msb.*

Operation:

```
lsb ← lsbminus32 + 32
msb ← msbminus32 + 32
if (lsb > msb) then
   UNPREDICTABLE
endif
GPR[rt] ← GPR[rt]63..msb+1 || GPR[rs]msb-lsb..0 || GPR[rt]lsb-1..0
```

Exceptions:

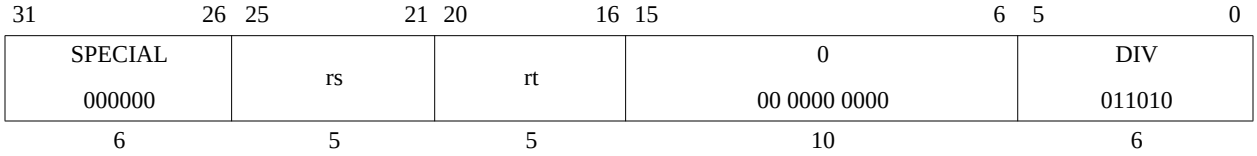
Reserved Instruction

Programming Notes

The assembler will accept any value of *pos* and *size* that satisfies the relationship 0 < pos+size ≤ 64 and emit DINS, DINSM, or DINSU as appropriate to the values. Programmers should always specify the DINS mnemonic and let the assembler select the instruction to use.

Divide Word

DIV



Format: `DIV rs, rt` **MIPS32**

Purpose:

To divide a 32-bit signed integers

Description: `(HI, LO) ← GPR[rs] / GPR[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.

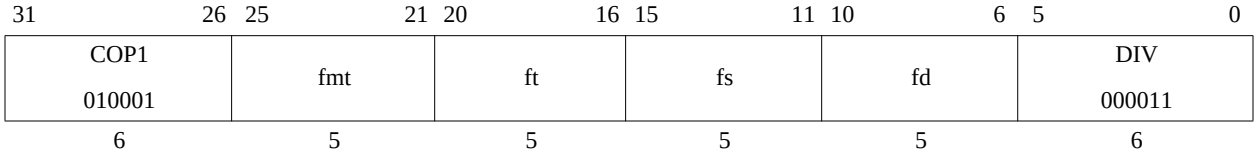
As an example, the C programming language in a UNIX® environment expects division by zero to either terminate the program or execute a program-specified signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test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

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.

Floating Point Divide DIV.fmt



Format: `DIV.S fd, fs, ft` **MIPS32**

`DIV.D fd, fs, ft` **MIPS32**

Purpose:

To divide FP values

Description: `FPR[fd] ← FPR[fs] / FPR[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 **UNPREDICABLE**.

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))
```

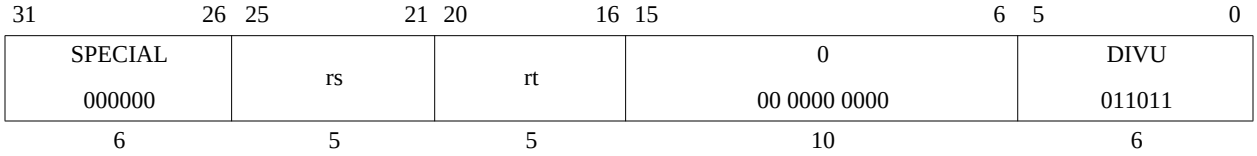
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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

Divide Unsigned Word DIVU



Format: `DIVU rs, rt` **MIPS32**

Purpose:

To divide a 32-bit unsigned integers

Description: `(HI, LO) ← GPR[rs] / GPR[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 **UNPREDICTABLE**.

Operation:

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

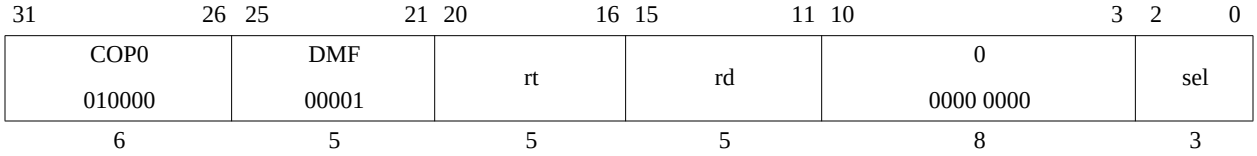
Programming Notes:

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.

Doubleword Move from Coprocessor 0 — DMFC0



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: `GPR[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:

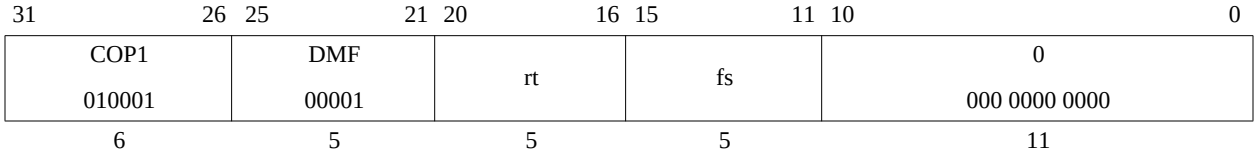
```
datadoubleword ← CPR[0,rd,sel]
GPR[rt] ← datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Doubleword Move from Floating Point — DMFC1



Format: `DMFC1 rt,fs` **MIPS64**

Purpose:

To move a doubleword from an FPR to a GPR.

Description: `GPR[rt] ← FPR[fs]`

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

Restrictions:

Operation:

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

Exceptions:

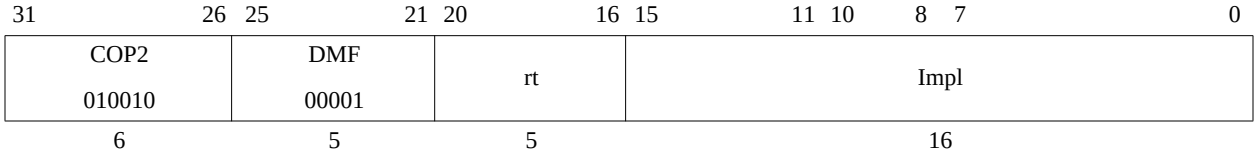
Coprocessor Unusable

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



Format: `DMFC2 rt, rd` **MIPS64**

`DMFC2, rt, rd,sel` **MIPS64**

The syntax shown above is an example using DMFC1 as a model. The specific syntax is implementation dependent.

Purpose:

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

Description: `GPR[rt] ← CP2CPR[Impl]`

The contents of the coprocessor 2 register denoted by the *Impl* field is loaded into GPR *rt*. The interpretation of the *Impl* field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:

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

Operation:

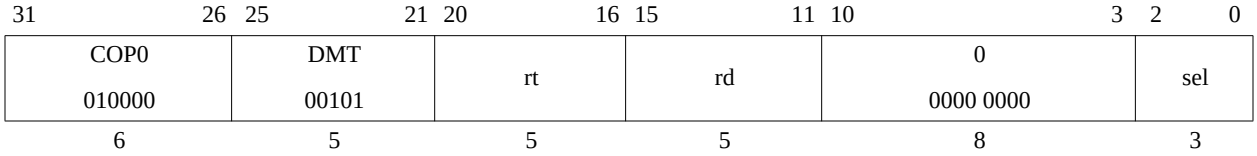
```
datadoubleword ← CP2CPR[Impl]
GPR[rt] ← datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Doubleword Move to Coprocessor 0 — DMTC0



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] ← GPR[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:

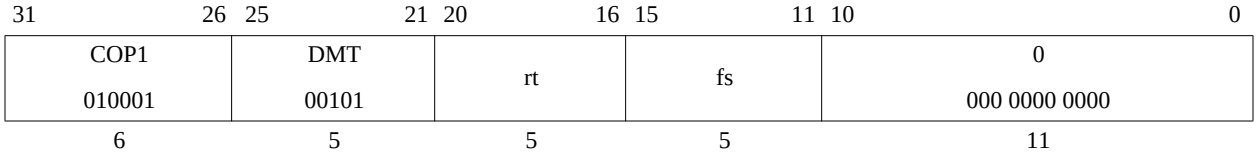
```
datadoubleword ← GPR[rt]
CPR[0,rd,sel] ← datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Doubleword Move to Floating Point — DMTC1



Format: `DMTC1 rt, fs` **MIPS64**

Purpose:

To copy a doubleword from a GPR to an FPR

Description: `FPR[fs] ← GPR[rt]`

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

Restrictions:

Operation:

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

Exceptions:

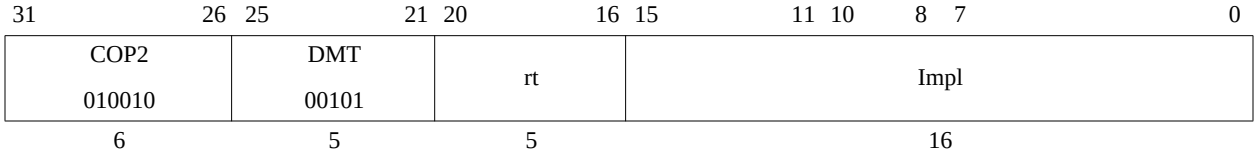
Coprocessor Unusable

Reserved Instruction

Historical Information:

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

Doubleword Move to Coprocessor 2 — DMTC2



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

`DMTC2 rt, rd, sel` **MIPS64**

The syntax shown above is an example using DMTC1 as a model. The specific syntax is implementation dependent.

Purpose:

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

Description: `CPR[2, rd, sel] ← GPR[rt]`

The contents GPR *rt* are loaded into the coprocessor 2 register denoted by the *Impl* field. The interpretation of the *Impl* field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:

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

Operation:

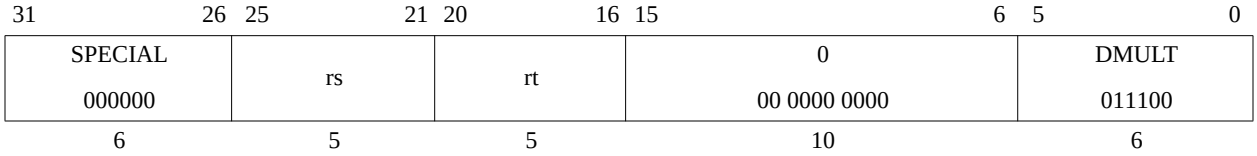
```
datadoubleword ← GPR[rt]
CP2CPR[Impl]← datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Doubleword Multiply DMULT



Format: `DMULT rs, rt` **MIPS64**

Purpose:

To multiply 64-bit signed integers

Description: `(LO, HI) ← GPR[rs] × GPR[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 ← prod63..0
HI ← prod127..64
```

Exceptions:

Reserved Instruction

Programming Notes:

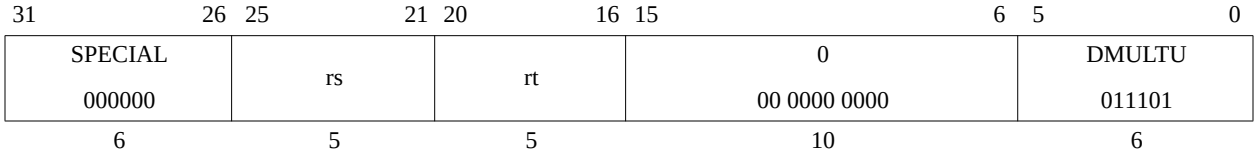
In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result, 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 Multiply Unsigned DMULTU



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

Purpose:

To multiply 64-bit unsigned integers

Description: `(LO, HI) ← GPR[rs] × GPR[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 ← prod63..0
HI ← prod127..64
```

Exceptions:

Reserved Instruction

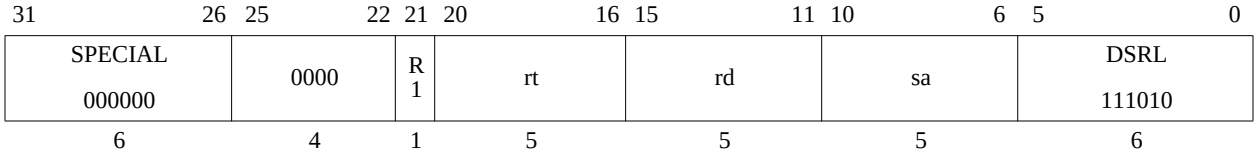
Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written 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.



Format: `DROTR rd, rt, sa` **MIPS64 Release 2**

Purpose:

To execute a logical right-rotate of a doubleword by a fixed amount—0 to 31 bits

Description: `GPR[rd] ← GPR[rt] ↔(right) sa`

The doubleword contents of GPR *rt* are rotated right; the result is placed in GPR *rd*. The bit-rotate amount in the range 0 to 31 is specified by *sa*.

Restrictions:

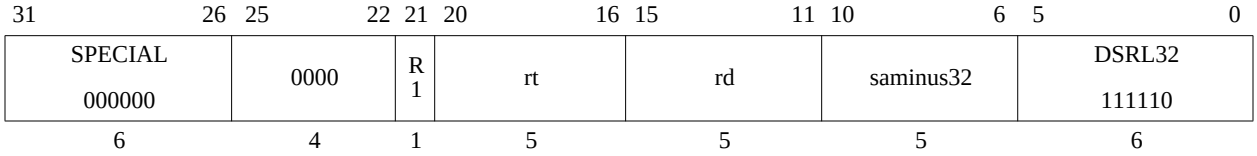
Operation:

```
s          ← 0 || sa
GPR[rd]    ← GPR[rt]s-1..0 || GPR[rt]63..s
```

Exceptions:

Reserved Instruction

Doubleword Rotate Right Plus 32 DROTR32



Format: `DROTR32 rd, rt, sa` **MIPS64 Release 2**

Purpose:

To execute a logical right-rotate of a doubleword by a fixed amount—32 to 63 bits

Description: `GPR[rd] ← GPR[rt] ↔(right) (saminus32+32)`

The 64-bit doubleword contents of GPR *rt* are rotated right; the result is placed in GPR *rd*. The bit-rotate amount in the range 32 to 63 is specified by *saminus32*+32.

Restrictions:

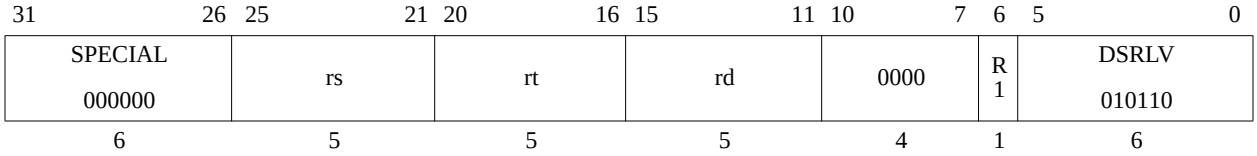
Operation:

```
s          ← 1 || sa      /* 32+saminus32 */
GPR[rd]    ← GPR[rt]s-1..0 || GPR[rt]63..s
```

Exceptions:

Reserved Instruction

Doubleword Rotate Right Variable DROTRV



Format: `DROTRV rd, rt, rs` **MIPS64 Release 2**

Purpose:

To execute a logical right-rotate of a doubleword by a variable number of bits

Description: `GPR[rd] ← GPR[rt] ↔(right) GPR[rs]`

The 64-bit doubleword contents of GPR *rt* are rotated right; the result is placed in GPR *rd*. The bit-rotate amount in the range 0 to 63 is specified by the low-order 6 bits in GPR *rs.*

Restrictions:

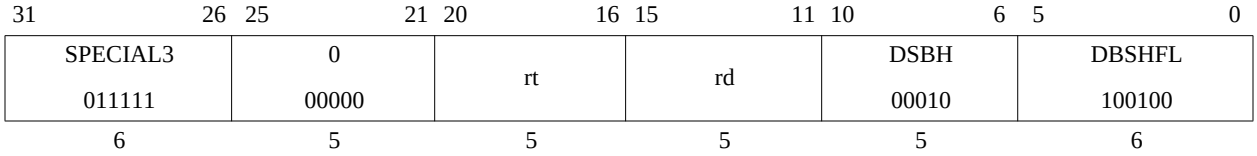
Operation:

```
s          ← GPR[rs]5..0
GPR[rd]    ← GPR[rt]s-1..0 || GPR[rt]63..s
```

Exceptions:

Reserved Instruction

Doubleword Swap Bytes Within Halfwords — DSBH



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL3	0	rt	rd	DSBH	DBSHFL
011111	00000			00010	100100
6	5	5	5	5	6

Format: `dsbh rd, rt` **MIPS64 Release 2**

Purpose:

To swap the bytes within each halfword of GPR *rt* and store the value into GPR *rd*.

Description: `GPR[rd] ← SwapBytesWithinHalfwords(GPR[rt])`

Within each halfword of GPR *rt* the bytes are swapped and stored in GPR *rd*.

Restrictions:

In implementations Release 1 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

$$\begin{aligned}\text{GPR[rd]} \leftarrow\ & \text{GPR[rt]}_{55..48} \parallel \text{GPR[rt]}_{63..56} \parallel \text{GPR[rt]}_{39..32} \parallel \text{GPR[rt]}_{47..40} \parallel \\ & \text{GPR[rt]}_{23..16} \parallel \text{GPR[rt]}_{31..24} \parallel \text{GPR[rt]}_{7..0} \parallel \text{GPR[rt]}_{15..8}\end{aligned}$$

Exceptions:

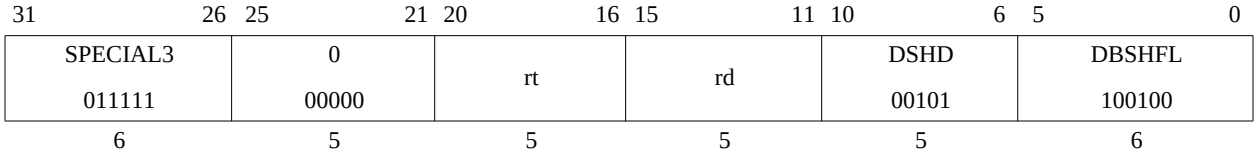
Reserved Instruction

Programming Notes:

The DSBH and DSHD instructions can be used to convert doubleword data of one endianness to the other endianness. For example:

```
ld     t0, 0(a1)           /* Read doubleword value */
dsbh   t0, t0              /* Convert endiannes of the halfwords */
dshd   t0, t0              /* Swap the halfwords within the doublewords */
```

Doubleword Swap Halfwords Within Doublewords — DSHD



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL3	0	rt	rd	DSHD	DBSHFL
011111	00000			00101	100100
6	5	5	5	5	6

Format: `dshd rd, rt` **MIPS64 Release 2**

Purpose:

To swap the halfwords of GPR *rt* and store the value into GPR *rd*.

Description: `GPR[rd] ← SwapHalfwordsWithinDoublewords(GPR[rt])`

The halfwords of GPR *rt* are swapped and stored in GPR *rd*.

Restrictions:

In implementations of Release 1 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

$GPR[rd] \leftarrow GPR[rt]_{15..0} \parallel GPR[rt]_{31..16} \parallel GPR[rt]_{47..32} \parallel GPR[rt]_{63..48}$

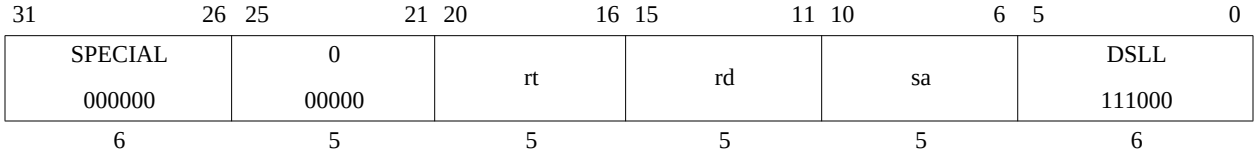
Exceptions:

Reserved Instruction

Programming Notes:

The DSBH and DSHD instructions can be used to convert doubleword data of one endianness to the other endianness. For example:

```
ld     t0, 0(a1)           /* Read doubleword value */
dsbh   t0, t0              /* Convert endiannes of the halfwords */
dshd   t0, t0              /* Swap the halfwords within the doublewords */
```



Format: `DSLL rd, rt, sa` **MIPS64**

Purpose:

To execute a left-shift of a doubleword by a fixed amount—0 to 31 bits

Description: `GPR[rd] ← GPR[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:

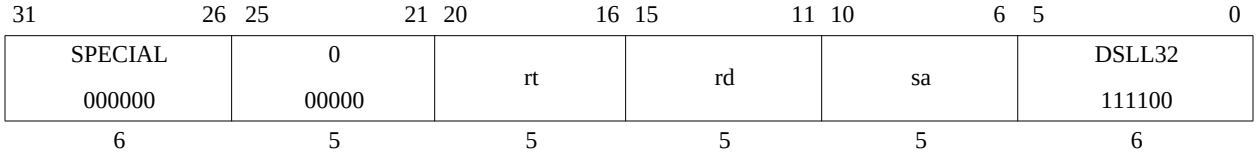
Operation:

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

Exceptions:

Reserved Instruction

Doubleword Shift Left Logical Plus 32 — DSLL32



Format: `DSLL32 rd, rt, sa` **MIPS64**

Purpose:

To execute a left-shift of a doubleword by a fixed amount—32 to 63 bits

Description: `GPR[rd] ← GPR[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:

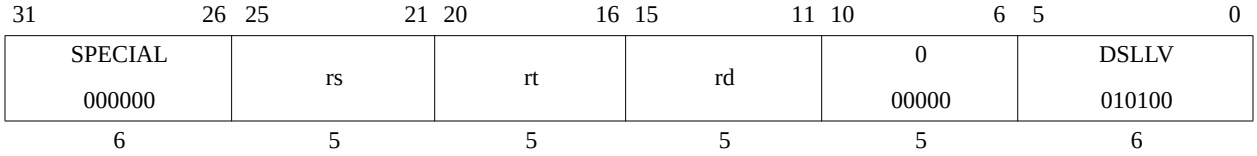
Operation:

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

Exceptions:

Reserved Instruction

Doubleword Shift Left Logical Variable DSLLV



Format: `DSLLV rd, rt, rs` **MIPS64**

Purpose:

To execute a left-shift of a doubleword by a variable number of bits

Description: `GPR[rd] ← GPR[rt] << GPR[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:

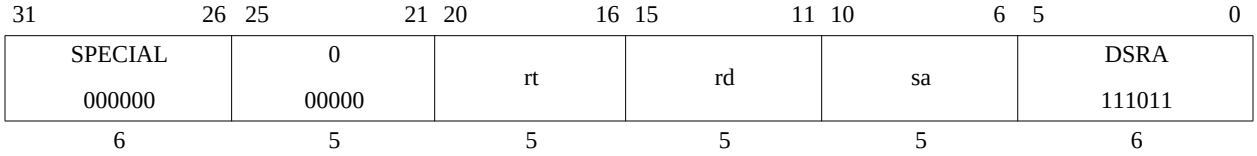
Operation:

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

Exceptions:

Reserved Instruction

Doubleword Shift Right Arithmetic DSRA



Format: `DSRA rd, rt, sa` **MIPS64**

Purpose:

To execute an arithmetic right-shift of a doubleword by a fixed amount—0 to 31 bits

Description: `GPR[rd] ← GPR[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:

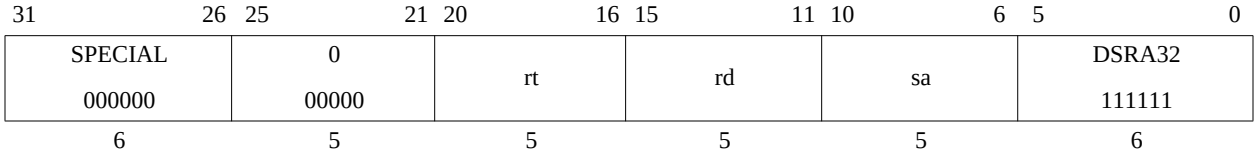
Operation:

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

Exceptions:

Reserved Instruction

Doubleword Shift Right Arithmetic Plus 32 — DSRA32



Format: `DSRA32 rd, rt, sa` **MIPS64**

Purpose:

To execute an arithmetic right-shift of a doubleword by a fixed amount—32 to 63 bits

Description: `GPR[rd] ← GPR[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:

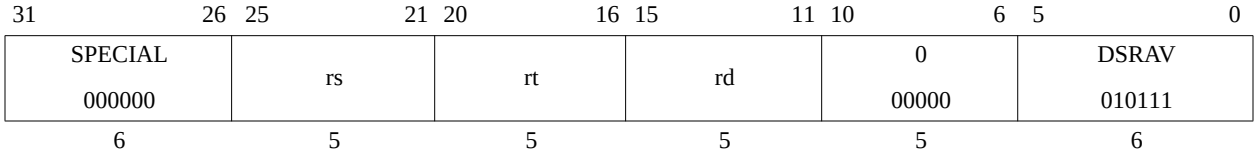
```
s       ← 1 || sa    /* 32+sa */
GPR[rd] ← (GPR[rt]63)^s || GPR[rt]63..s
```

Exceptions:

Reserved Instruction

Doubleword Shift Right Arithmetic Variable

DSRAV



Format: `DSRAV rd, rt, rs` **MIPS64**

Purpose:

To execute an arithmetic right-shift of a doubleword by a variable number of bits

Description: `GPR[rd] ← GPR[rt] >> GPR[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:

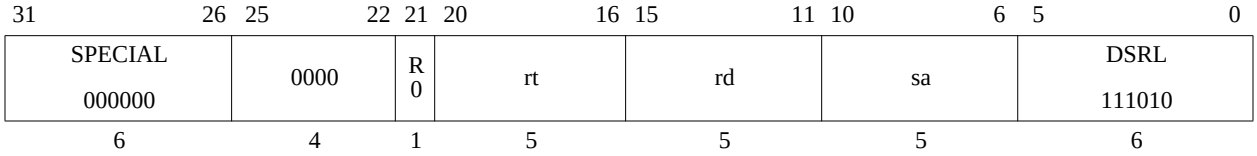
Operation:

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

Exceptions:

Reserved Instruction

Doubleword Shift Right Logical DSRL



Format: `DSRL rd, rt, sa` **MIPS64**

Purpose:

To execute a logical right-shift of a doubleword by a fixed amount—0 to 31 bits

Description: `GPR[rd] ← GPR[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:

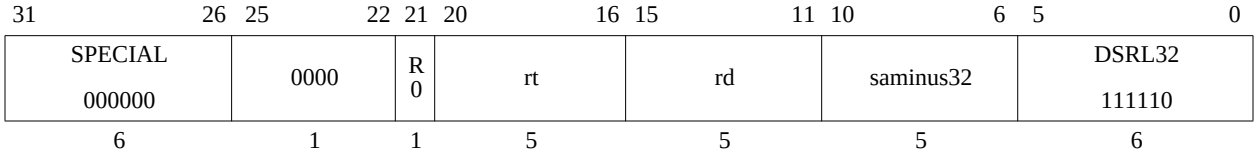
Operation:

```
s          ← 0 || sa
GPR[rd]    ← 0^s || GPR[rt]_63..s
```

Exceptions:

Reserved Instruction

Doubleword Shift Right Logical Plus 32 — DSRL32



Format: `DSRL32 rd, rt, sa` **MIPS64**

Purpose:

To execute a logical right-shift of a doubleword by a fixed amount—32 to 63 bits

Description: `GPR[rd] ← GPR[rt] >> (saminus32+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 *saminus32*+32.

Restrictions:

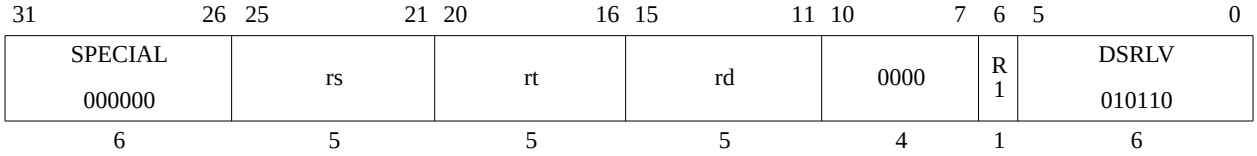
Operation:

```
s          ← 1 || sa    /* 32+saminus32 */
GPR[rd]    ← 0^s || GPR[rt]_63..s
```

Exceptions:

Reserved Instruction

Doubleword Shift Right Logical Variable DSRLV



Format: `DSRLV rd, rt, rs` **MIPS64**

Purpose:

To execute a logical right-shift of a doubleword by a variable number of bits

Description: `GPR[rd] ← GPR[rt] >> GPR[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:

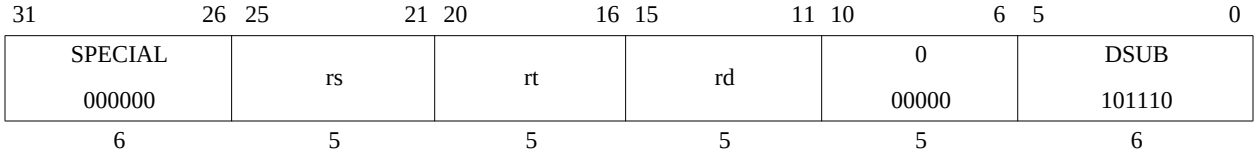
Operation:

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

Exceptions:

Reserved Instruction

Doubleword Subtract DSUB



Format: `DSUB rd, rs, rt` **MIPS64**

Purpose:

To subtract 64-bit integers; trap on overflow

Description: `GPR[rd] ← GPR[rs] - GPR[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
```

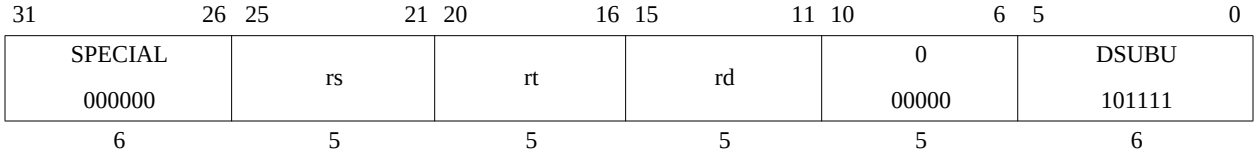
Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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

Doubleword Subtract Unsigned DSUBU



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	DSUBU 101111
6	5	5	5	5	6

Format: `DSUBU rd, rs, rt` **MIPS64**

Purpose:

To subtract 64-bit integers

Description: `GPR[rd] ← GPR[rs] - GPR[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]
```

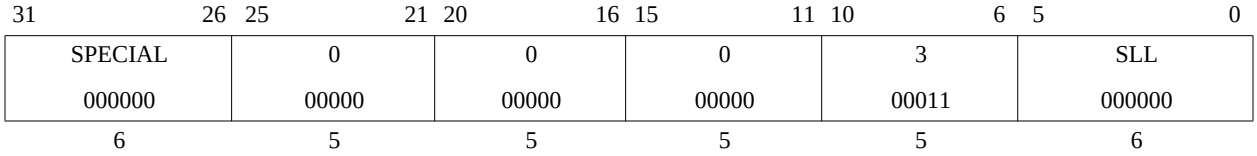
Exceptions:

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.

Execution Hazard Barrier EHB



Format: `EHB` **MIPS32 Release 2**

Purpose:

To stop instruction execution until all execution hazards have been cleared.

Description:

EHB is the assembly idiom used to denote execution hazard barrier. The actual instruction is interpreted by the hardware as SLL r0, r0, 3.

This instruction alters the instruction issue behavior on a pipelined processor by stopping execution until all execution hazards have been cleared. Other than those that might be created as a consequence of setting $Status_{CU0}$, there are no execution hazards visible to an unprivileged program running in User Mode. All execution hazards created by previous instructions are cleared for instructions executed immediately following the EHB, even if the EHB is executed in the delay slot of a branch or jump. The EHB instruction does not clear instruction hazards - such hazards are cleared by the JALR.HB, JR.HB, and ERET instructions.

Restrictions:

None

Operation:

```
ClearExecutionHazards()
```

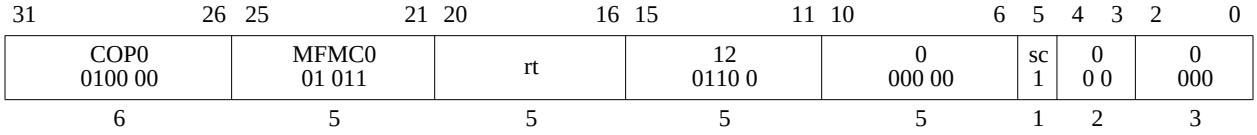
Exceptions:

None

Programming Notes:

In MIPS64 Release 2 implementations, this instruction resolves all execution hazards. On a superscalar processor, EHB alters the instruction issue behavior in a manner identical to SSNOP. For backward compatibility with Release 1 implementations, the last of a sequence of SSNOPs can be replaced by an EHB. In Release 1 implementations, the EHB will be treated as an SSNOP, thereby preserving the semantics of the sequence. In Release 2 implementations, replacing the final SSNOP with an EHB should have no performance effect because a properly sized sequence of SSNOPs will have already cleared the hazard. As EHB becomes the standard in MIPS implementations, the previous SSNOPs can be removed, leaving only the EHB.

Enable Interrupts EI



Format: `EI` **MIPS32 Release 2**

`EI rt` **MIPS32 Release 2**

Purpose:

To return the previous value of the *Status* register and enable interrupts. If EI is specified without an argument, GPR r0 is implied, which discards the previous value of the Status register.

Description: `GPR[rt] ← Status; Status`$_{IE}$ `← 1`

The current value of the *Status* register is sign-extended and loaded into general register rt. The Interrupt Enable (IE) bit in the *Status* register is then set.

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

This operation specification is for the general interrupt enable/disable operation, with the *sc* field as a variable. The individual instructions DI and EI have a specific value for the *sc* field.

```
data ← Status
GPR[rt] ← sign_extend(data)
Status_IE ← 1
```

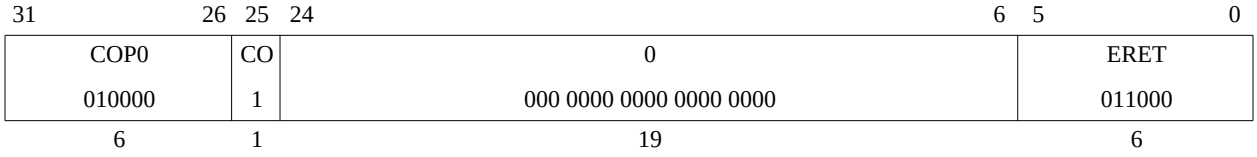
Exceptions:

Coprocessor Unusable
Reserved Instruction (Release 1 implementations)

Programming Notes:

The effects of this instruction are identical to those accomplished by the sequence of reading *Status* into a GPR, setting the IE bit, and writing the result back to *Status*. Unlike the multiple instruction sequence, however, the EI instruction can not be aborted in the middle by an interrupt or exception.

This instruction creates an execution hazard between the change to the Status register and the point where the change to the interrupt enable takes effect. This hazard is cleared by the EHB, JALR.HB, JR.HB, or ERET instructions. Software must not assume that a fixed latency will clear the execution hazard.



Format: `ERET` **MIPS32**

Purpose:

To return from interrupt, exception, or error trap.

Description:

ERET clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ in a Release 2 implementation, and returns to the interrupted instruction at the completion of interrupt, exception, or error 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 that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the ERET returns.

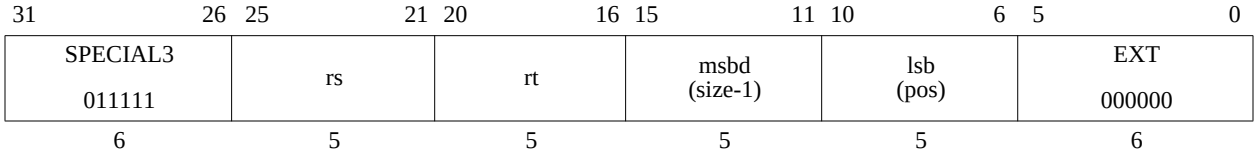
In a Release 2 implementation, ERET does not restore $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ if $Status_{BEV} = 1$, or if $Status_{ERL} = 1$ because any exception that sets $Status_{ERL}$ to 1 (Reset, Soft Reset, NMI, or cache error) does not save $SRSCtl_{CSS}$ in $SRSCtl_{PSS}$. If software sets $Status_{ERL}$ to 1, it must be aware of the operation of an ERET that may be subsequently executed.

Operation:

```
if Status_ERL = 1 then
    temp ← ErrorEPC
    Status_ERL ← 0
else
    temp ← EPC
    Status_EXL ← 0
    if (ArchitectureRevision ≥ 2) and (SRSCtl_HSS > 0) and (Status_BEV = 0)then
        SRSCtl_CSS ← SRSCtl_PSS
    endif
endif
if IsMIPS16Implemented() then
    PC ← temp_63..1 || 0
    ISAMode ← temp_0
else
    PC ← temp
endif
LLbit ← 0
ClearHazards()
```

Exceptions:

Coprocessor Unusable Exception



Format: `ext rt, rs, pos, size` **MIPS32 Release 2**

Purpose:

To extract a bit field from GPR *rs* and store it right-justified into GPR *rt*.

Description: `GPR[rt] ← ExtractField(GPR[rs], msbd, lsb)`

The bit field starting at bit *pos* and extending for *size* bits is extracted from GPR *rs* and stored zero-extended and right-justified in GPR *rt*. The assembly language arguments *pos* and *size* are converted by the assembler to the instruction fields *msbd* (the most significant bit of the destination field in GPR *rt*), in instruction bits 15..11, and *lsb* (least significant bit of the source field in GPR *rs*), in instruction bits 10..6, as follows:

```
msbd ← size-1
lsb ← pos
```

The values of *pos* and *size* must satisfy all of the following relations:

```
0 ≤ pos < 32
0 < size ≤ 32
0 < pos+size ≤ 32
```

Figure 3-9 shows the symbolic operation of the instruction.

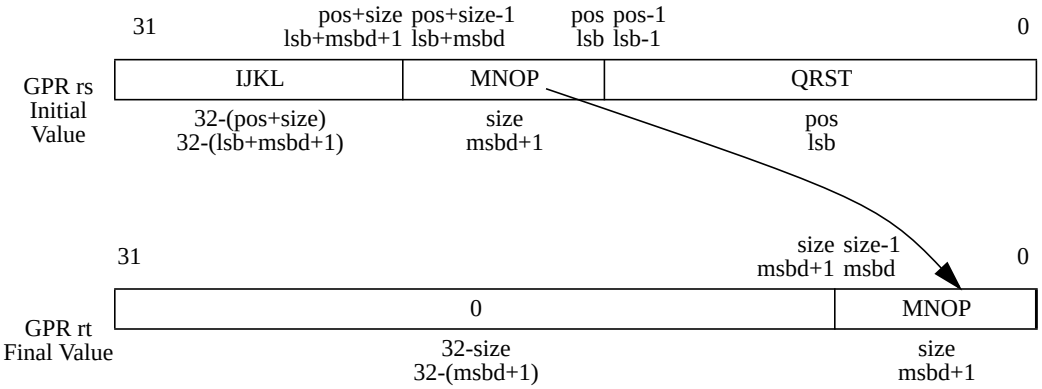


Figure 3-9 Operation of the EXT Instruction

Restrictions:

In implementations prior to Release of the architecture, this instruction resulted in a Reserved Instruction Exception.

The operation is **UNPREDICTABLE** if *lsb*+*msbd* > 31.

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

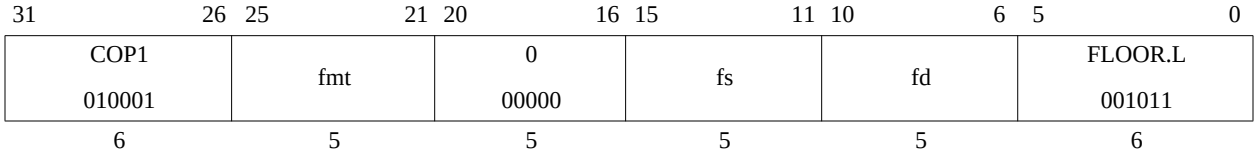
Operation:

```
if ((lsb + msbd) > 31) or (NotWordValue(GPR[rs])) then
   UNPREDICTABLE
endif
temp ← sign_extend(0^(32-(msbd+1)) || GPR[rs]_(msbd+lsb..lsb))
GPR[rt] ← temp
```

Exceptions:

Reserved Instruction

Floating Point Floor Convert to Long Fixed Point — FLOOR.L.fmt



Format: `FLOOR.L.S fd, fs` **MIPS64, MIPS32 Release 2**

`FLOOR.L.D fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

To convert an FP value to 64-bit fixed point, rounding down

Description: `FPR[fd] ← convert_and_round(FPR[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))
```

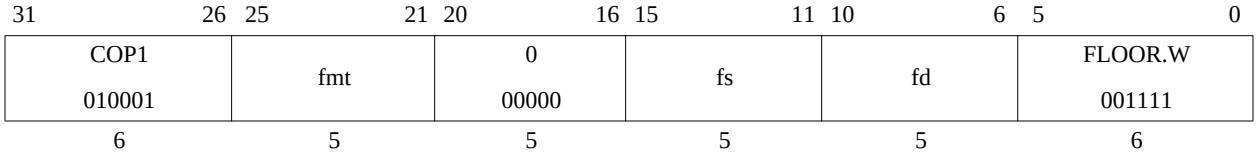
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow

Floating Point Floor Convert to Word Fixed Point — FLOOR.W.fmt



Format: `FLOOR.W.S fd, fs` **MIPS32**
`FLOOR.W.D fd, fs` **MIPS32**

Purpose:

To convert an FP value to 32-bit fixed point, rounding down

Description: `FPR[fd] ← convert_and_round(FPR[fs])`

The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word 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^{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:

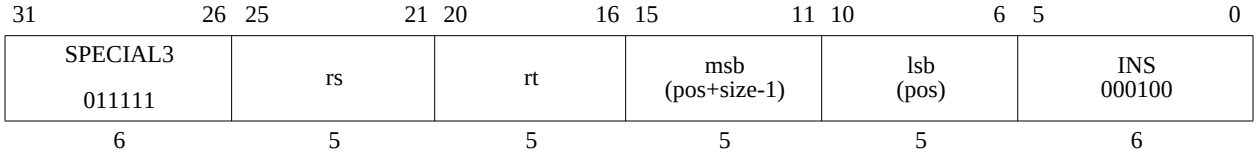
```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow



Format: `ins rt, rs, pos, size` **MIPS32 Release 2**

Purpose:

To merge a right-justified bit field from GPR *rs* into a specified field in GPR *rt*.

Description: `GPR[rt] ← InsertField(GPR[rt], GPR[rs], msb, lsb)`

The right-most *size* bits from GPR *rs* are merged into the value from GPR *rt* starting at bit position *pos*. The result is placed back in GPR *rt*. The assembly language arguments *pos* and *size* are converted by the assembler to the instruction fields *msb* (the most significant bit of the field), in instruction bits 15..11, and *lsb* (least significant bit of the field), in instruction bits 10..6, as follows:

```
msb ← pos+size-1
lsb ← pos
```

The values of *pos* and *size* must satisfy all of the following relations:

```
0 ≤ pos < 32
0 < size ≤ 32
0 < pos+size ≤ 32
```

Figure 3-10 shows the symbolic operation of the instruction.

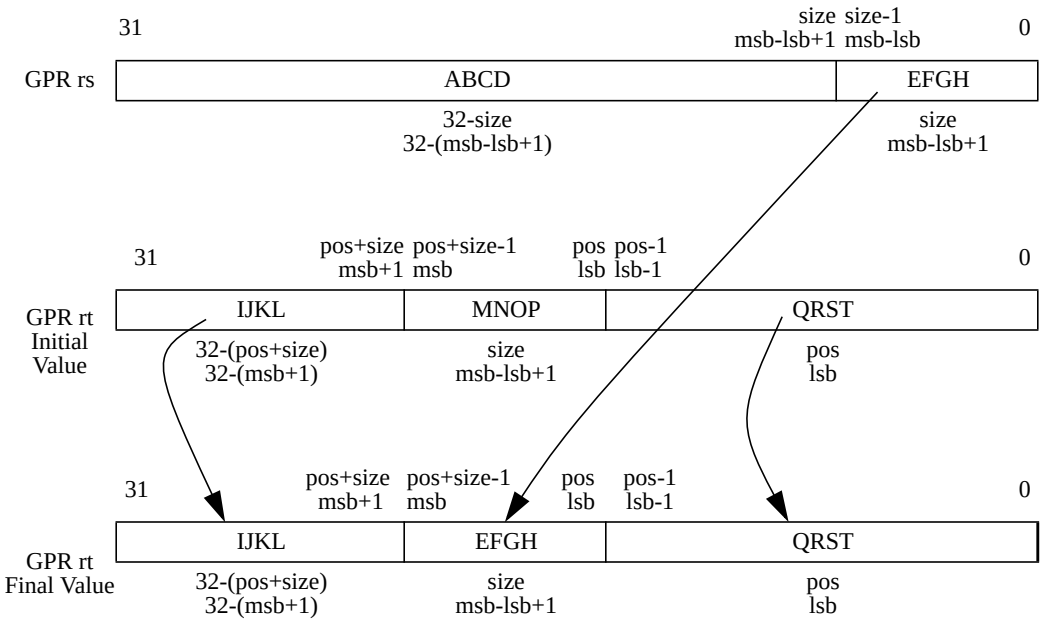


Figure 3-10 Operation of the INS Instruction

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

The operation is **UNPREDICTABLE** if *lsb* > *msb.*

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

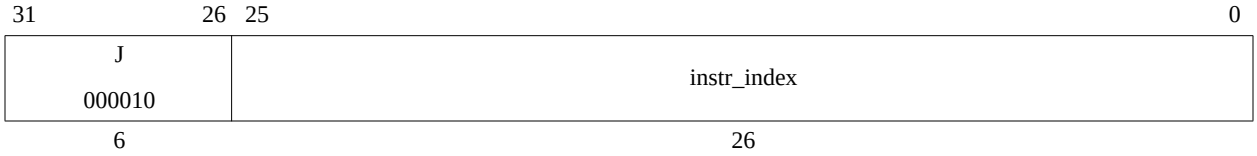
Operation:

```
    if (lsb > msb) or (NotWordValue(GPR[rs])) or (NotWordValue(GPR[rt]))) then
       UNPREDICTABLE
    endif
GPR[rt] ← sign_extend(GPR[rt]31..msb+1 || GPR[rs]msb-lsb..0 || GPR[rt]lsb-1..0)
```

Exceptions:

Reserved Instruction

Jump J



Format: `J target` **MIPS32**

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-1..28 || instr_index || 0^2
```

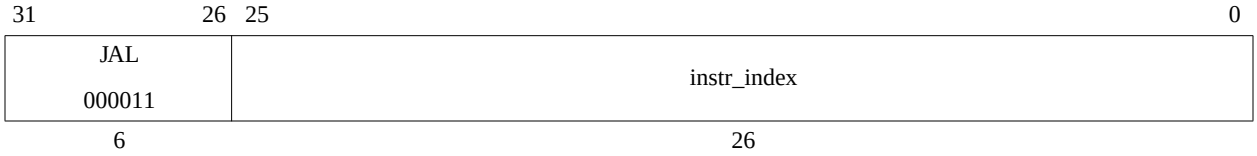
Exceptions:

None

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch 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.



Format: `JAL target` **MIPS32**

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-1..28 || instr_index || 0^2
```

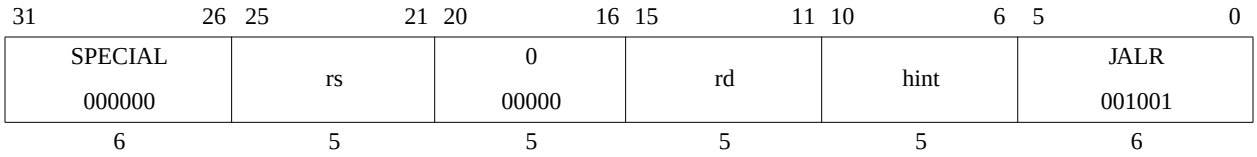
Exceptions:

None

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch 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.



Format: `JALR rs (rd = 31 implied)` **MIPS32**

`JALR rd, rs` **MIPS32**

Purpose:

To execute a procedure call to an instruction address in a register

Description: `GPR[rd] ← return_addr, PC ← GPR[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 MIPS16e 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 MIPS16e 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. 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

In release 1 of the architecture, the only defined hint field value is 0, which sets default handling of JALR. In Release 2 of the architecture, bit 10 of the hint field is used to encode a hazard barrier. See the JALR.HB instruction description for additional information.

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 **UNPREDICTABLE**. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16e 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 MIPS16e 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:

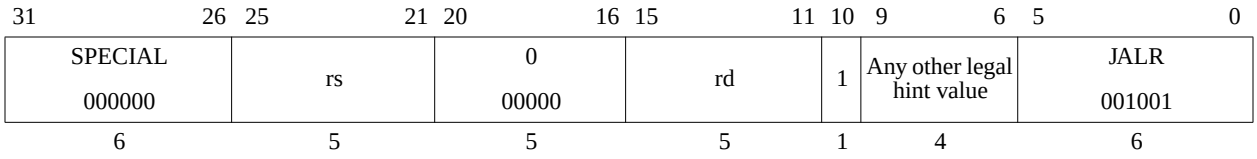
```
I:  temp ← GPR[rs]
    GPR[rd] ← PC + 8
I+1:if Config1CA = 0 then
        PC ← temp
    else
        PC ← tempGPRLEN-1..1 || 0
        ISAMode ← temp0
    endif
```

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.



Format: `JALR.HB rs (rd = 31 implied)` **MIPS32 Release 2**

`JALR.HB rd, rs` **MIPS32 Release 2**

Purpose:

To execute a procedure call to an instruction address in a register and clear all execution and instruction hazards

Description: `GPR[rd] ← return_addr, PC ← GPR[rs], clear execution and instruction hazards`

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*. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself. 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

JALR.HB implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the JALR.HB instruction jumps. An equivalent barrier is also implemented by the ERET instruction, but that instruction is only available if access to Coprocessor 0 is enabled, whereas JALR.HB is legal in all operating modes.

This instruction clears both execution and instruction hazards. Refer to the EHB instruction description for the method of clearing execution hazards alone.

JALR.HB uses bit 10 of the instruction (the upper bit of the hint field) to denote the hazard barrier operation.

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 **UNPREDICTABLE**. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

The effective target address in GPR *rs* must be naturally-aligned. 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.

Restrictions, cont.:

After modifying an instruction stream mapping or writing to the instruction stream, execution of those instructions has **UNPREDICTABLE** behavior until the instruction hazard has been cleared with JALR.HB, JR.HB, ERET, or DERET. Further, the operation is **UNPREDICTABLE** if the mapping of the current instruction stream is modified.

JALR.HB does not clear hazards created by any instruction that is executed in the delay slot of the JALR.HB. Only hazards created by instructions executed before the JALR.HB are cleared by the JALR.HB.

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 Config1CA = 0 then
        PC ← temp
    else
        PC ← tempGPRLEN-1..1 || 0
        ISAMode ← temp0
    endif
    ClearHazards()
```

Exceptions:

None

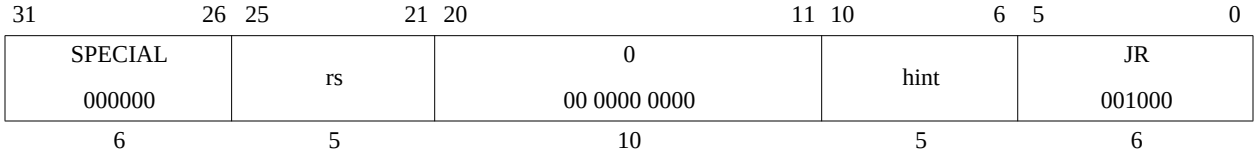
Programming Notes:

JALR and JALR.HB are the only branch-and-link instructions 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.

This instruction implements the final step in clearing execution and instruction hazards before execution continues. A hazard is created when a Coprocessor 0 or TLB write affects execution or the mapping of the instruction stream, or after a write to the instruction stream. When such a situation exists, software must explicitly indicate to hardware that the hazard should be cleared. Execution hazards alone can be cleared with the EHB instruction. Instruction hazards can only be cleared with a JR.HB, JALR.HB, or ERET instruction. These instructions cause hardware to clear the hazard before the instruction at the target of the jump is fetched. Note that because these instructions are encoded as jumps, the process of clearing an instruction hazard can often be included as part of a call (JALR) or return (JR) sequence, by simply replacing the original instructions with the HB equivalent.

Example: Clearing hazards due to an ASID change

```
/*
 * Code used to modify ASID and call a routine with the new
 * mapping established.
 *
 * a0 = New ASID to establish
 * a1 = Address of the routine to call
 */
   mfc0   v0, C0_EntryHi      /* Read current ASID */
   li     v1, ~M_EntryHiASID  /* Get negative mask for field */
   and    v0, v0, v1          /* Clear out current ASID value */
   or     v0, v0, a0          /* OR in new ASID value */
   mtc0   v0, C0_EntryHi      /* Rewrite EntryHi with new ASID */
   jalr.hb a1                 /* Call routine, clearing the hazard */
   nop
```



Format: `JR rs` **MIPS32**

Purpose:

To execute a branch to an instruction address in a register

Description: `PC ← GPR[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 MIPS16e 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 MIPS16e 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 MIPS16e 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.

In release 1 of the architecture, the only defined hint field value is 0, which sets default handling of JR. In Release 2 of the architecture, bit 10 of the hint field is used to encode an instruction hazard barrier. See the JR.HB instruction description for additional information.

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 Config1CA = 0 then
        PC ← temp
    else
        PC ← tempGPRLEN-1..1 || 0
        ISAMode ← temp0
    endif
```

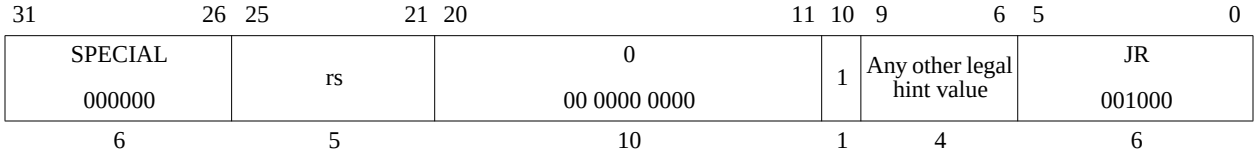
Exceptions:

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.

Jump Register with Hazard Barrier — JR.HB



Format: `JR.HB rs` **MIPS32 Release 2**

Purpose:

To execute a branch to an instruction address in a register and clear all execution and instruction hazards.

Description: `PC ← GPR[rs], clear execution and instruction hazards`

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

JR.HB implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the JR.HB instruction jumps. An equivalent barrier is also implemented by the ERET instruction, but that instruction is only available if access to Coprocessor 0 is enabled, whereas JR.HB is legal in all operating modes.

This instruction clears both execution and instruction hazards. Refer to the EHB instruction description for the method of clearing execution hazards alone.

JR.HB uses bit 10 of the instruction (the upper bit of the hint field) to denote the hazard barrier operation.

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.

After modifying an instruction stream mapping or writing to the instruction stream, execution of those instructions has **UNPREDICTABLE** behavior until the hazard has been cleared with JALR.HB, JR.HB, ERET, or DERET. Further, the operation is **UNPREDICTABLE** if the mapping of the current instruction stream is modified.

JR.HB does not clear hazards created by any instruction that is executed in the delay slot of the JALR.HB. Only hazards created by instructions executed before the JR.HB are cleared by the JALR.HB.

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 Config1CA = 0 then
       PC ← temp
    else
       PC ← tempGPRLEN-1..1 || 0
       ISAMode ← temp0
    endif
    ClearHazards()
```

Exceptions:

None

Programming Notes:

This instruction implements the final step in clearing execution and instruction hazards before execution continues. A hazard is created when a Coprocessor 0 or TLB write affects execution or the mapping of the instruction stream, or after a write to the instruction stream. When such a situation exists, software must explicitly indicate to hardware that the hazard should be cleared. Execution hazards alone can be cleared with the EHB instruction. Instruction hazards can only be cleared with a JR.HB, JALR.HB, or ERET instruction. These instructions cause hardware to clear the hazard before the instruction at the target of the jump is fetched. Note that because these instructions are encoded as jumps, the process of clearing an instruction hazard can often be included as part of a call (JALR) or return (JR) sequence, by simply replacing the original instructions with the HB equivalent.

Example: Clearing hazards due to an ASID change

```
/*
 * Routine called to modify ASID and return with the new
 * mapping established.
 *
 * a0 = New ASID to establish
 */
   mfc0   v0, C0_EntryHi      /* Read current ASID */
   li     v1, ~M_EntryHiASID  /* Get negative mask for field */
   and    v0, v0, v1          /* Clear out current ASID value */
   or     v0, v0, a0          /* OR in new ASID value */
   mtc0   v0, C0_EntryHi      /* Rewrite EntryHi with new ASID */
   jr.hb  ra                  /* Return, clearing the hazard */
   nop
```

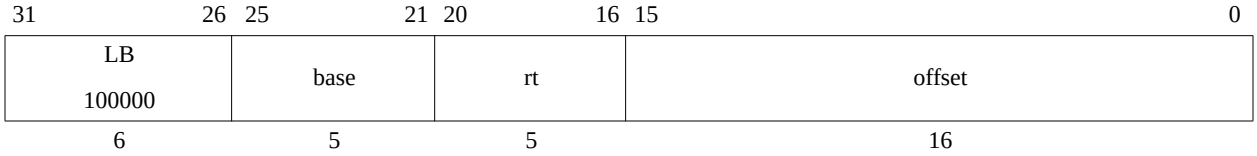
Example: Making a write to the instruction stream visible

```
/*
 * Routine called after new instructions are written to
 * make them visible and return with the hazards cleared.
 */
   {Synchronize the caches - see the SYNCI and CACHE instructions}
   sync                       /* Force memory synchronization */
   jr.hb  ra                  /* Return, clearing the hazard */
   nop
```

Example: Clearing instruction hazards in-line

```
    la     AT, 10f
    jr.hb  AT                  /* Jump to next instruction, clearing */
    nop                        /*   hazards */
10:
```

Load Byte LB



Format: `LB rt, offset(base)` **MIPS32**

Purpose:

To load a byte from memory as a signed value

Description: `GPR[rt] ← memory[GPR[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

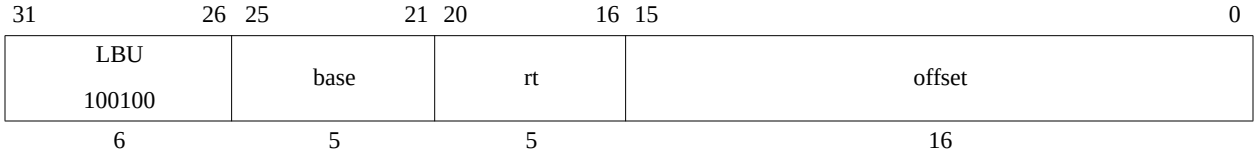
Operation:

```
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, Watch

Load Byte Unsigned — LBU



Format: `LBU rt, offset(base)` **MIPS32**

Purpose:

To load a byte from memory as an unsigned value

Description: `GPR[rt] ← memory[GPR[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:

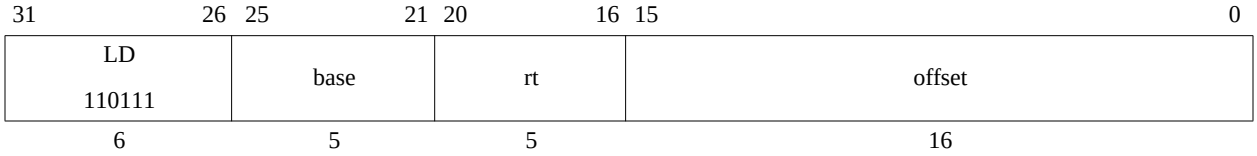
```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor ReverseEndian^3)
memdoubleword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte   ← vAddr_2..0 xor BigEndianCPU^3
GPR[rt]← zero_extend(memdoubleword_7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

Load Doubleword

LD



Format: `LD rt, offset(base)` **MIPS64**

Purpose:

To load a doubleword from memory

Description: `GPR[rt] ← memory[GPR[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.

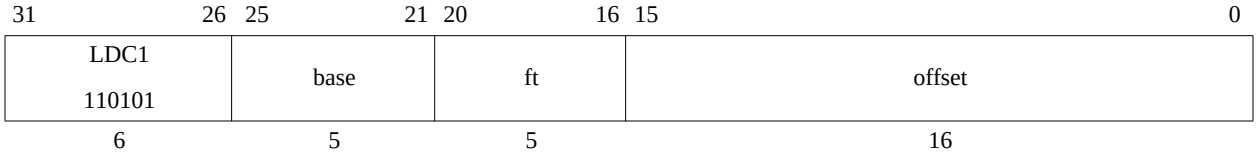
Operation:

```
vAddr         ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 0^3 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, Watch

Load Doubleword to Floating Point — LDC1



Format: `LDC1 ft, offset(base)` **MIPS32**

Purpose:

To load a doubleword from memory to an FPR

Description: `FPR[ft] ← memory[GPR[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:

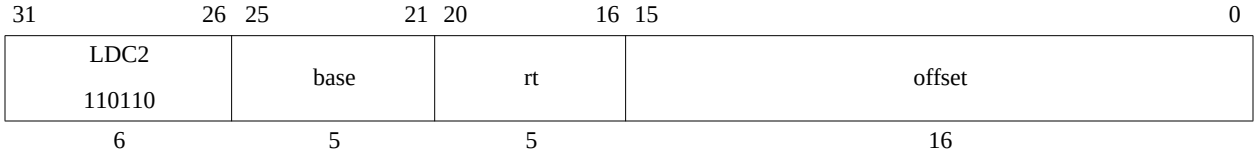
```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 0^3 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)

StoreFPR(ft, UNINTERPRETED_DOUBLEWORD, memdoubleword)
```

Exceptions:

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

Load Doubleword to Coprocessor 2 — LDC2



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

Purpose:

To load a doubleword from memory to a Coprocessor 2 register

Description: `CPR[2,rt,0] ← memory[GPR[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:

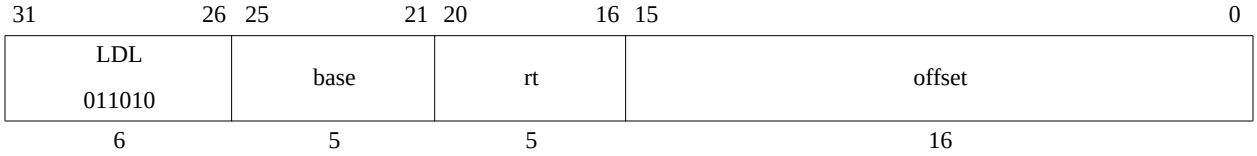
An Address Error exception occurs if $EffectiveAddress_{2..0} \neq 0$ (not doubleword-aligned).

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)
CPR[2,rt,0] ← memdoubleword
```

Exceptions:

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



Format: `LDL rt, offset(base)` **MIPS64**

Purpose:

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

Description: `GPR[rt] ← GPR[rt] MERGE memory[GPR[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 3-11 Unaligned Doubleword Load Using LDL and LDR

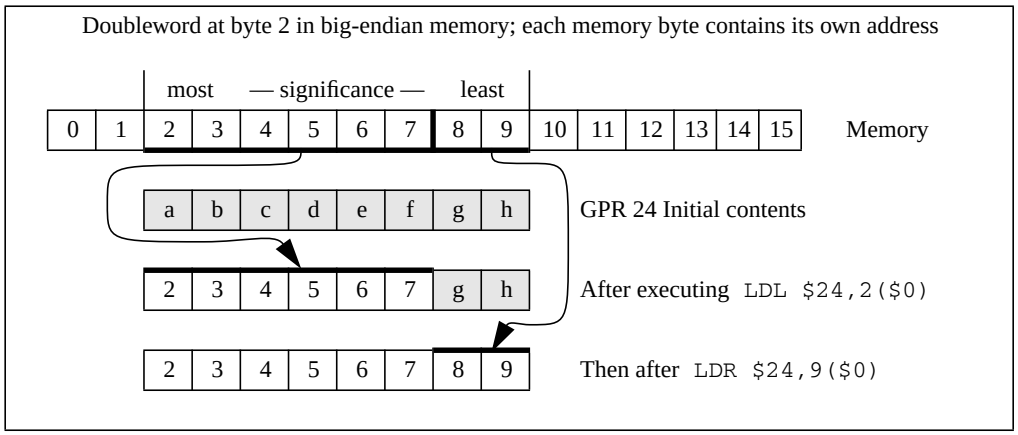
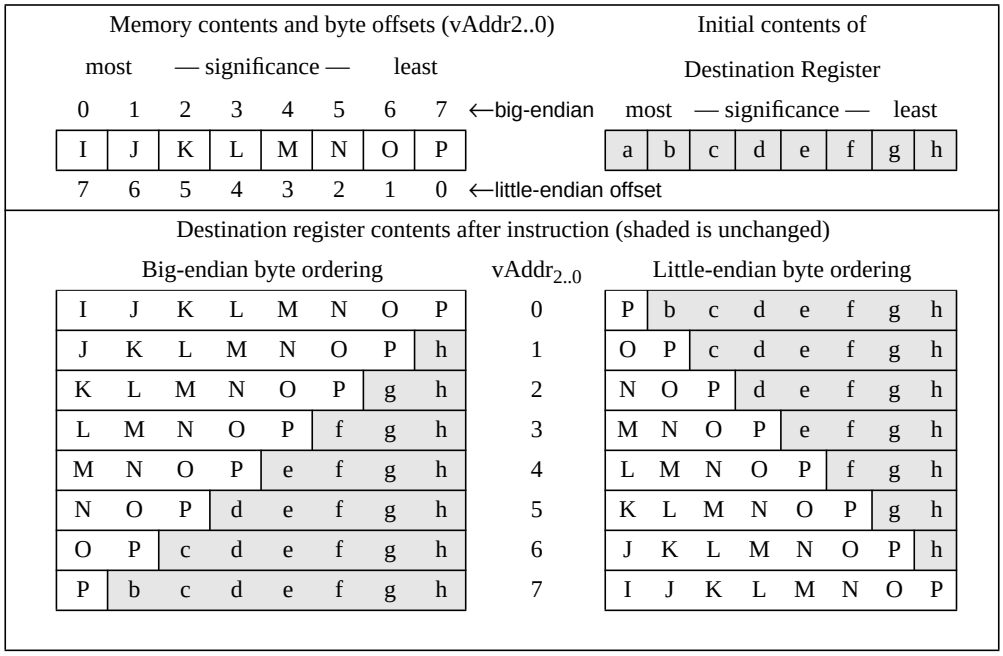


Figure 3-11 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 6 bytes, is located in the aligned doubleword starting with the most-significant byte at 2. LDL first loads these 6 bytes into the 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 3-12 shows the bytes loaded for every combination of offset and byte ordering.

Figure 3-12 Bytes Loaded by LDL Instruction



Restrictions:

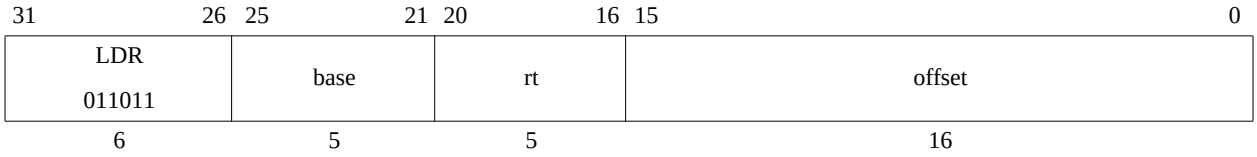
Operation:

```
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, Watch

Load Doubleword Right LDR



Format: `LDR rt, offset(base)` **MIPS64**

Purpose:

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

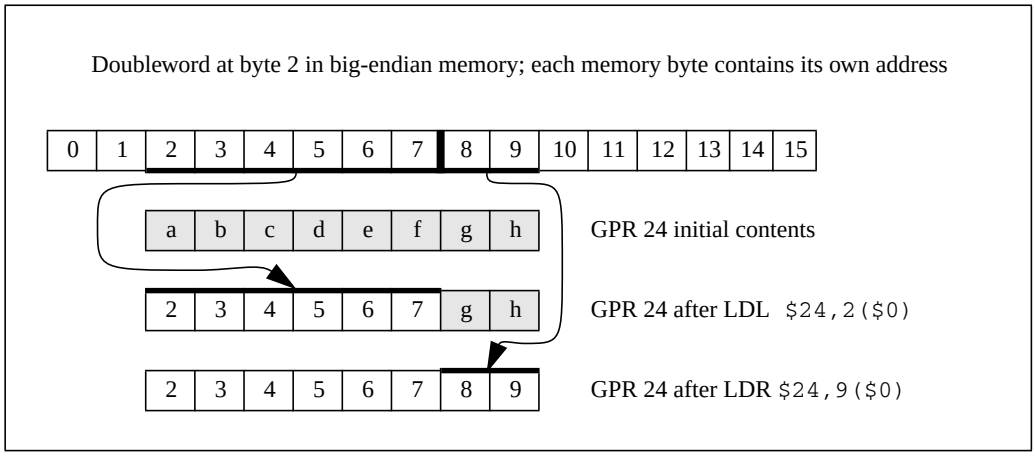
Description: `GPR[rt] ← GPR[rt] MERGE memory[GPR[base] + offset]`

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 3-13 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.

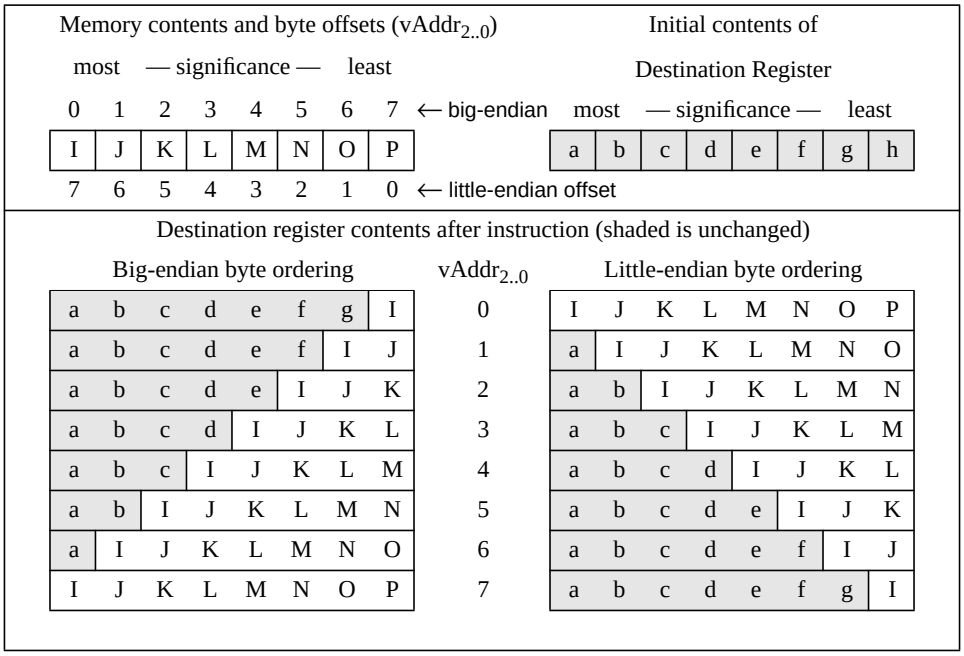
Figure 3-13 Unaligned Doubleword Load Using LDR and LDL



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 3-14 shows the bytes loaded for every combination of offset and byte ordering.

Figure 3-14 Bytes Loaded by LDR Instruction



Restrictions:

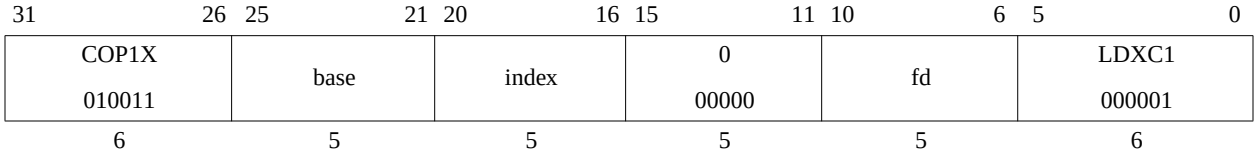
Operation: 64-bit processors

```
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, Watch

Load Doubleword Indexed to Floating Point LDXC1



Format: `LDXC1 fd, index(base)` **MIPS64**
MIPS32 Release 2

Purpose:

To load a doubleword from memory to an FPR (GPR+GPR addressing)

Description: `FPR[fd] ← memory[GPR[base] + GPR[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$_{2..0} \neq 0$ (not doubleword-aligned).

Operation:

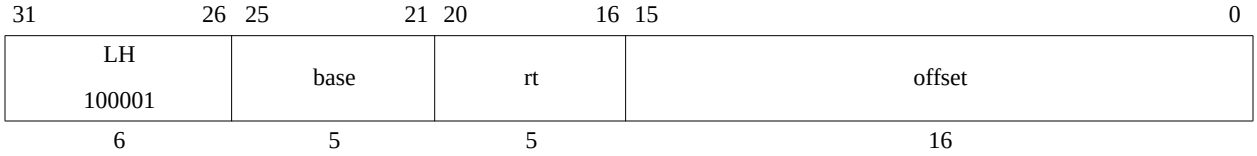
```
vAddr ← GPR[base] + GPR[index]
if vAddr2..0 ≠ 03 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
StoreFPR(ft, UNINTERPRETED_DOUBLEWORD, memdoubleword)
```

Exceptions:

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

Load Halfword

LH



Format: `LH rt, offset(base)` **MIPS32**

Purpose:

To load a halfword from memory as a signed value

Description: `GPR[rt] ← memory[GPR[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.

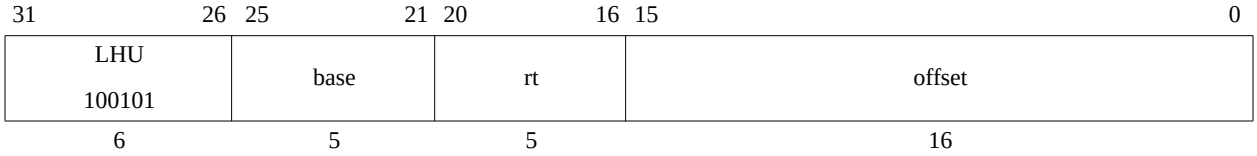
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-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, Watch

Load Halfword Unsigned LHU



Format: `LHU rt, offset(base)` **MIPS32**

Purpose:

To load a halfword from memory as an unsigned value

Description: `GPR[rt] ← memory[GPR[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:

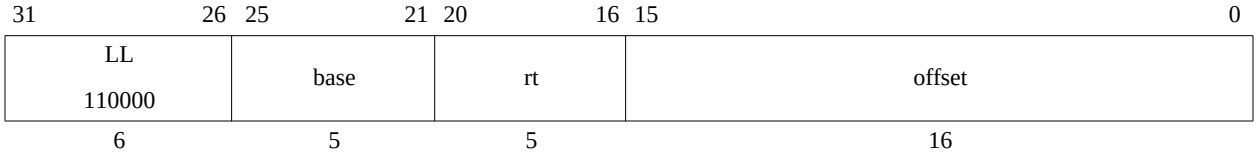
The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-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, Watch



Format: `LL rt, offset(base)` **MIPS32**

Purpose:

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

Description: `GPR[rt] ← memory[GPR[base] + offset]`

The LL and SC instructions provide the primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

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

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 synchronizable by all processors and I/O devices sharing the location; if it is not, the result in **UNPREDICTABLE**. Which storage is synchronizable is a function of both CPU and system implementations. See the documentation of the SC instruction for the formal definition.

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 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
memdoubleword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte  ← vAddr2..0 xor (BigEndianCPU || 0^2)
GPR[rt] ← sign_extend(memdoubleword31+8*byte..8*byte)
LLbit ← 1
```

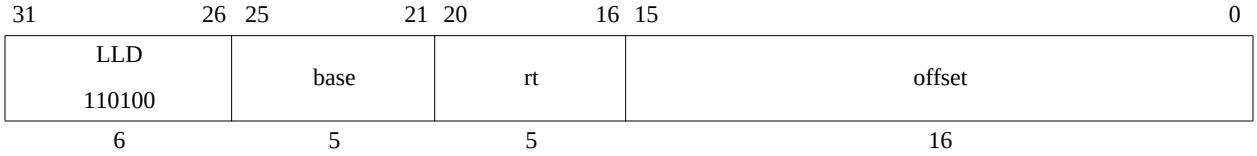
Exceptions:

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

Programming Notes:

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

Load Linked Doubleword LLD



Format: `LLD rt, offset(base)` **MIPS64**

Purpose:

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

Description: `GPR[rt] ← memory[GPR[base] + offset]`

The LLD and SCD instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

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

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 synchronizable by all processors and I/O devices sharing the location; if it is not, the result in **UNPREDICTABLE**. Which storage is synchronizable is a function of both CPU and system implementations. See the documentation of the SCD instruction for the formal definition.

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.

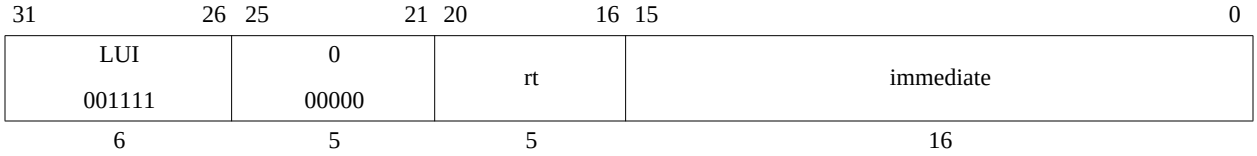
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 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, Watch

Load Upper Immediate LUI



Format: `LUI rt, immediate` **MIPS32**

Purpose:

To load a constant into the upper half of a word

Description: `GPR[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

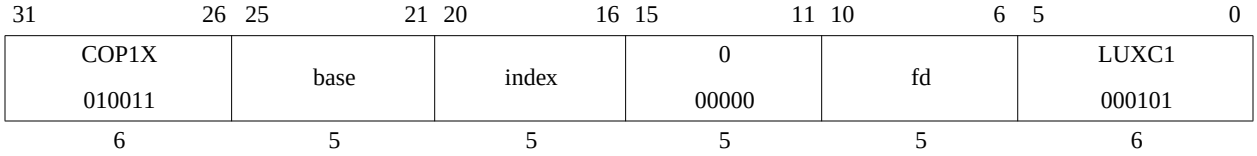
Operation:

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

Exceptions:

None

Load Doubleword Indexed Unaligned to Floating Point — LUXC1



Format: `LUXC1 fd, index(base)` **MIPS64**
MIPS32 Release 2

Purpose:

To load a doubleword from memory to an FPR (GPR+GPR addressing), ignoring alignment

Description: `FPR[fd] ← memory[(GPR[base] + GPR[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 FPR *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 **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

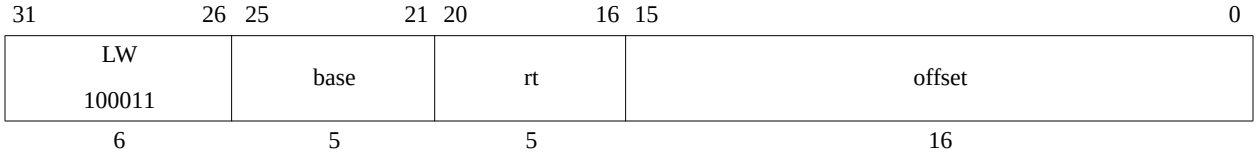
```
vAddr ← (GPR[base]+GPR[index])63..3 || 03
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)

StoreFPR(ft, UNINTERPRETED_DOUBLEWORD, memdoubleword)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, Watch

Load Word LW



Format: `LW rt, offset(base)` **MIPS32**

Purpose:

To load a word from memory as a signed value

Description: `GPR[rt] ← memory[GPR[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.

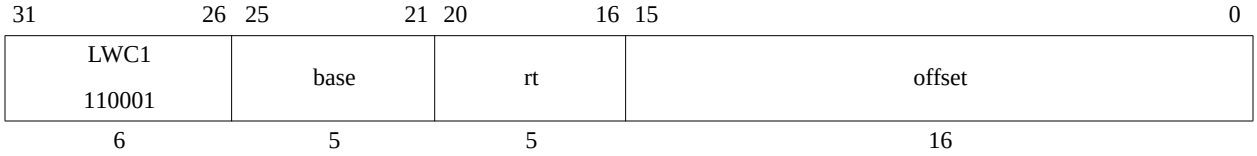
Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr2..0 xor (BigEndianCPU || 0^2)
GPR[rt]← sign_extend(memdoublewordbyte+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

Load Word to Floating Point — LWC1



Format: `LWC1 ft, offset(base)` **MIPS32**

Purpose:

To load a word from memory to an FPR

Description: `FPR[ft] ← memory[GPR[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 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 $EffectiveAddress_{1..0} \neq 0$ (not word-aligned).

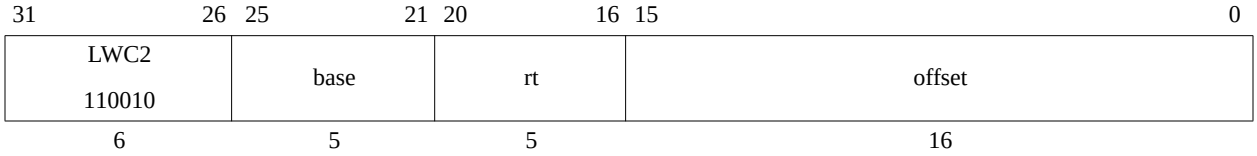
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
memdoubleword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
bytesel ← vAddr2..0 xor (BigEndianCPU || 0^2)
StoreFPR(ft, UNINTERPRETED_WORD,
          sign_extend(memdoublewordPSIZE-1+8*bytesel..8*bytesel))
```

Exceptions:

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

Load Word to Coprocessor 2 — LWC2



Format: `LWC2 rt, offset(base)` **MIPS32**

Purpose:

To load a word from memory to a COP2 register

Description: `CPR[2,rt,0] ← memory[GPR[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).

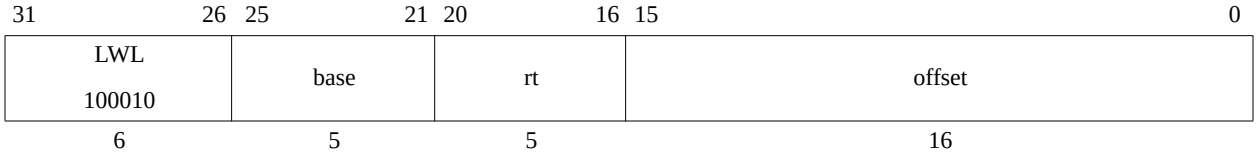
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr12..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
bytesel ← vAddr2..0 xor (BigEndianCPU || 0^2)
CPR[2,rt,0] ← sign_extend(memdoubleword31+8*bytesel..8*bytesel)
```

Exceptions:

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

Load Word Left LWL



Format: `LWL rt, offset(base)` **MIPS32**

Purpose:

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

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

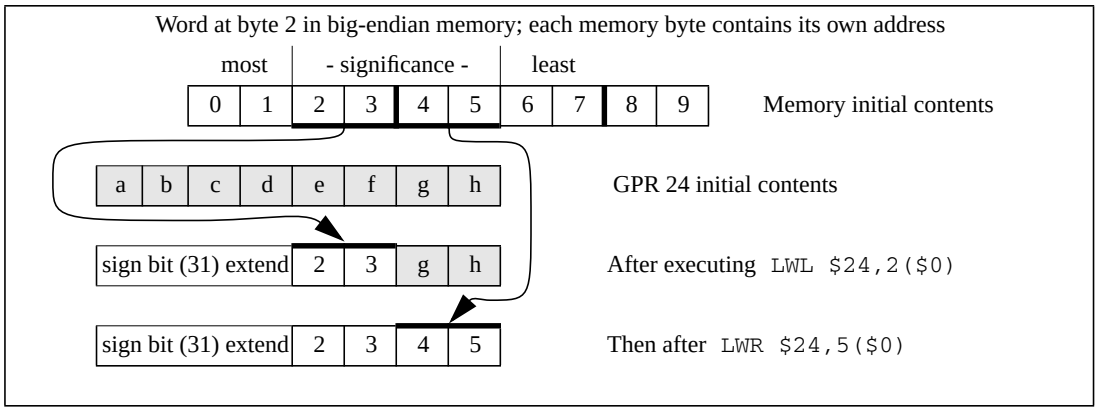
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

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.

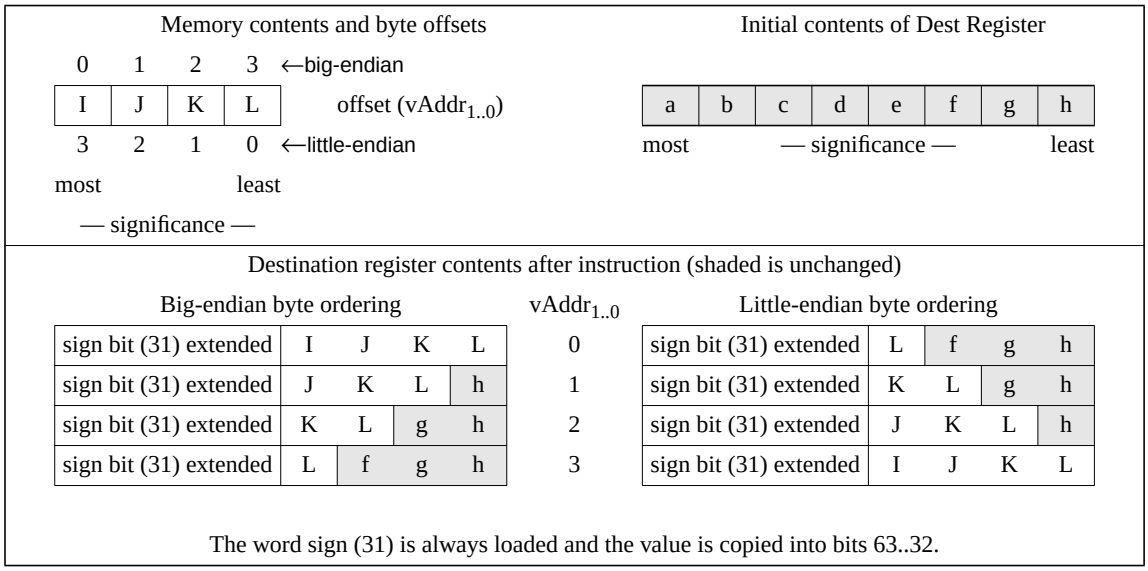
The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W,* 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2 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 3-15 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 3-16 Bytes Loaded by LWL Instruction



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   ← 0 || (vAddr_1..0 xor BigEndianCPU^2)
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, Watch

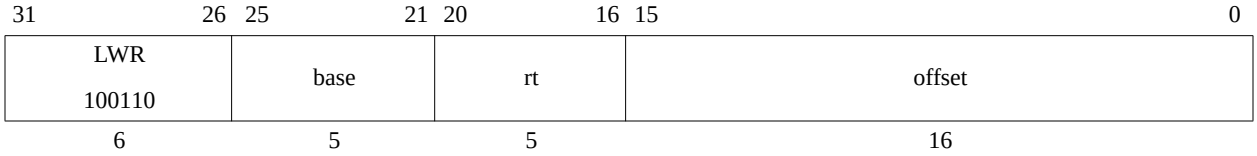
Programming Notes:

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.

Load Word Right LWR



Format: `LWR rt, offset(base)` **MIPS32**

Purpose:

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

Description: `GPR[rt] ← GPR[rt] MERGE memory[GPR[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.

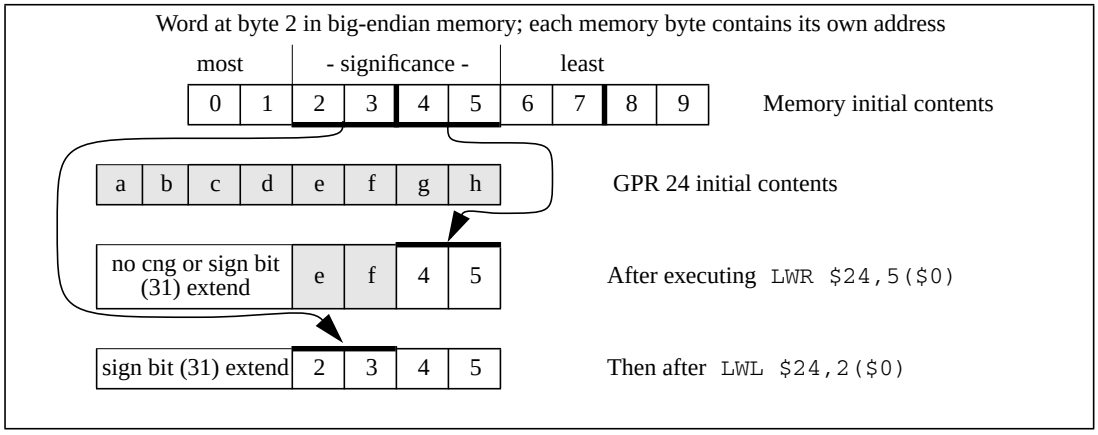
A part of *W*, the least-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. This part of *W* is loaded into the least-significant (right) part of the word in GPR *rt*. The remaining most-significant part of the word in GPR *rt* is unchanged.

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.

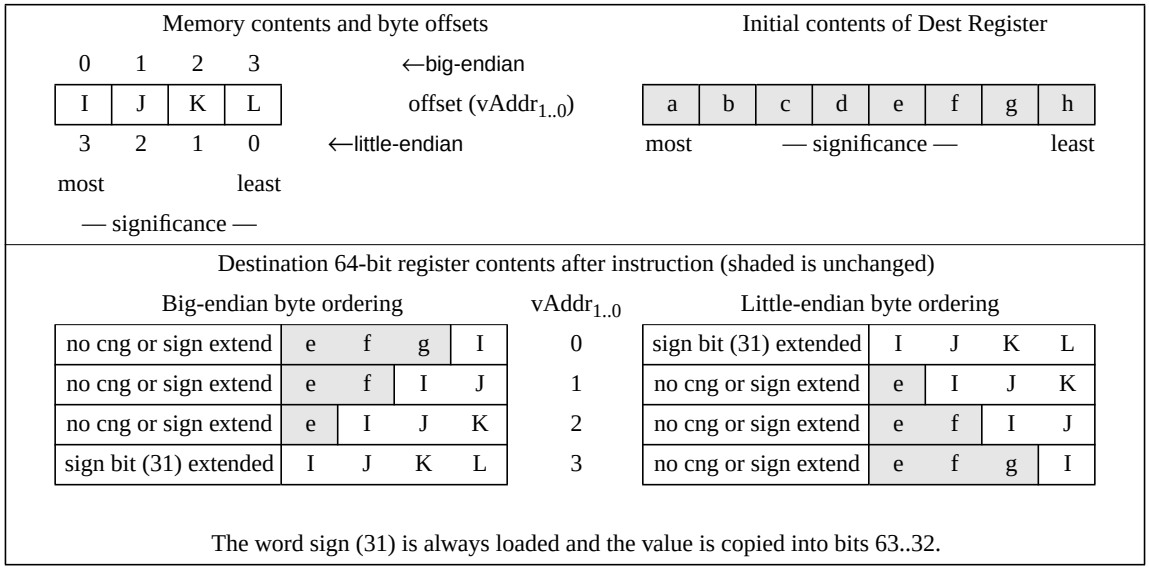
The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the least-significant byte at 5. First, LWR loads these 2 bytes into the right part of the destination register. Next, the complementary LWL loads the remainder of the unaligned word.

Figure 3-17 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 3-18 Bytes Loaded by LWR Instruction



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, Watch

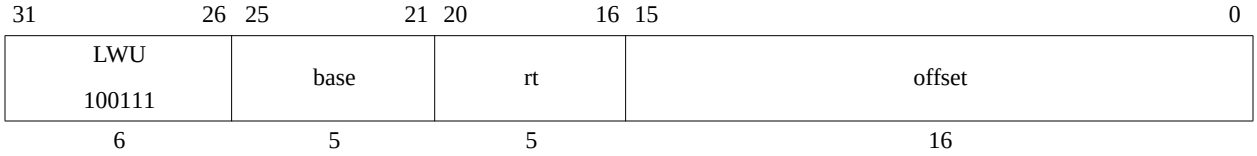
Programming Notes:

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.

Load Word Unsigned LWU



Format: `LWU rt, offset(base)` **MIPS64**

Purpose:

To load a word from memory as an unsigned value

Description: `GPR[rt] ← memory[GPR[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:

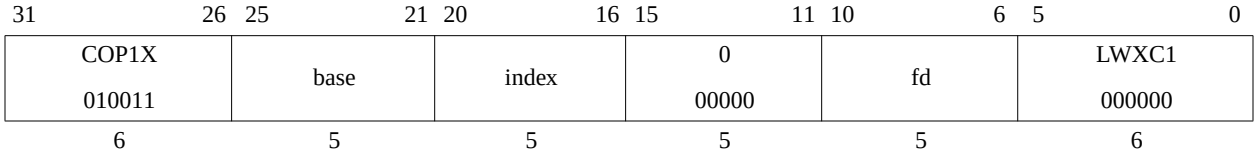
The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr2..0 xor (BigEndianCPU || 02)
GPR[rt]← 032 || memdoublewordBIT_POS[31+8*byte..8*byte]
```

Exceptions:

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



Format: `LWXC1 fd, index(base)` **MIPS64**
MIPS32 Release 2

Purpose:

To load a word from memory to an FPR (GPR+GPR addressing)

Description: `FPR[fd] ← memory[GPR[base] + GPR[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 FPR *fd*. The contents of GPR *index* and GPR *base* are added to form the effective address.

Restrictions:

An Address Error exception occurs if $EffectiveAddress_{1..0} \neq 0$ (not word-aligned).

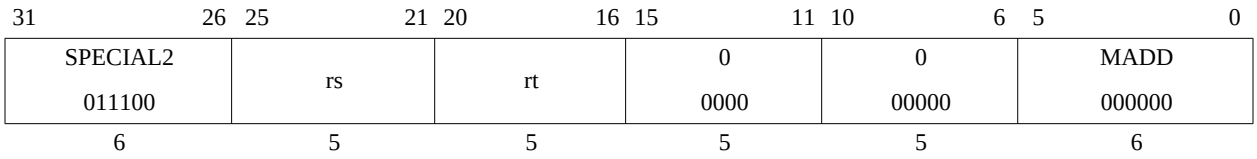
Operation:

```
vAddr ← GPR[base] + GPR[index]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
memdoubleword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
bytesel ← vAddr2..0 xor (BigEndianCPU || 0^2)
StoreFPR(ft, UNINTERPRETED_WORD,
          sign_extend(memdoublewords31+8*bytesel..8*bytesel))
```

Exceptions:

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

Multiply and Add Word to Hi,Lo — MADD



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: `(HI,LO) ← (HI,LO) + (GPR[rs] × GPR[rt])`

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:

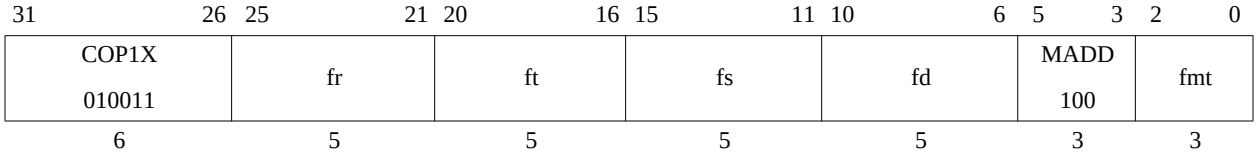
```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← (HI31..0 || LO31..0) + (GPR[rs]31..0 × GPR[rt]31..0)
HI ← sign_extend(temp63..32)
LO ← sign_extend(temp31..0)
```

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.



Format: `MADD.S fd, fr, fs, ft` **MIPS64, MIPS32 Release 2**

`MADD.D fd, fr, fs, ft` **MIPS64, MIPS32 Release 2**

`MADD.PS fd, fr, fs, ft` **MIPS64, MIPS32 Release 2**

Purpose:

To perform a combined multiply-then-add of FP values

Description: `FPR[fd] ← (FPR[fs] × FPR[ft]) + FPR[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 **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 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, (vfs ×fmt vft) +fmt vfr)
```

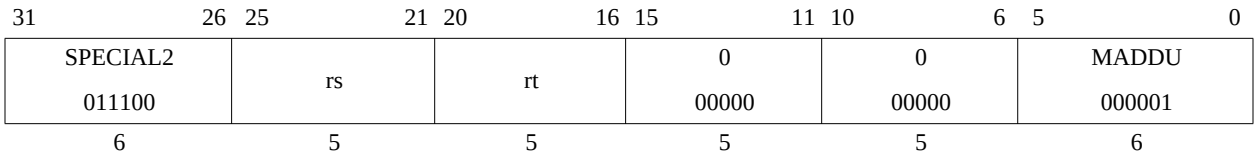
Exceptions:

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	rs	rt	0	0	MADDU
011100			00000	00000	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: `(HI,LO) ← (HI,LO) + (GPR[rs] × GPR[rt])`

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 ← (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)
```

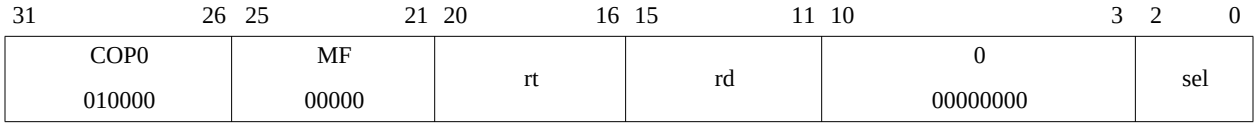
Exceptions:

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.

Move from Coprocessor 0 MFC0



Format: `MFC0 rt, rd` **MIPS32**

`MFC0 rt, rd, sel` **MIPS32**

Purpose:

To move the contents of a coprocessor 0 register to a general register.

Description: `GPR[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:

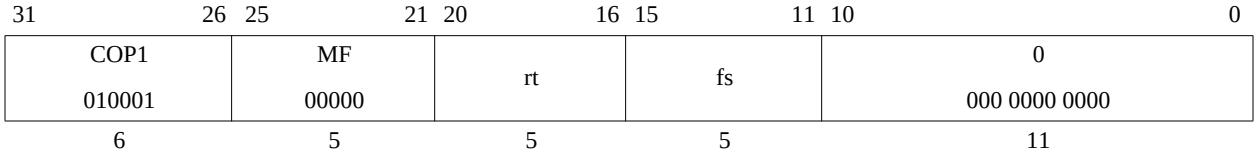
```
data ← CPR[0,rd,sel]31..0
GPR[rt] ← sign_extend(data)
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Move Word From Floating Point — MFC1



Format: `MFC1 rt, fs` **MIPS32**

Purpose:

To copy a word from an FPU (CP1) general register to a GPR

Description: `GPR[rt] ← FPR[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)
```

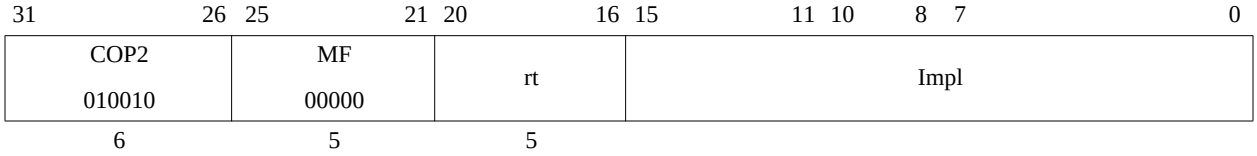
Exceptions:

Coprocessor Unusable, Reserved Instruction

Historical Information:

For MIPS I, MIPS II, and MIPS III the contents of GPR *rt* are **UNPREDICTABLE** for the instruction immediately following MFC1.

Move Word From Coprocessor 2 — MFC2



Format: `MFC2 rt, rd` **MIPS32**

`MFC2, rt, rd, sel` **MIPS32**

The syntax shown above is an example using MFC1 as a model. The specific syntax is implementation dependent.

Purpose:

To copy a word from a COP2 general register to a GPR

Description: `GPR[rt] ← CP2CPR[Impl]`

The contents of the coprocessor 2 register denoted by the *Impl* field are sign-extended and placed into general register rt. The interpretation of the *Impl* field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:

The results are **UNPREDICTABLE** if *Impl* specifies a coprocessor 2 register that does not exist.

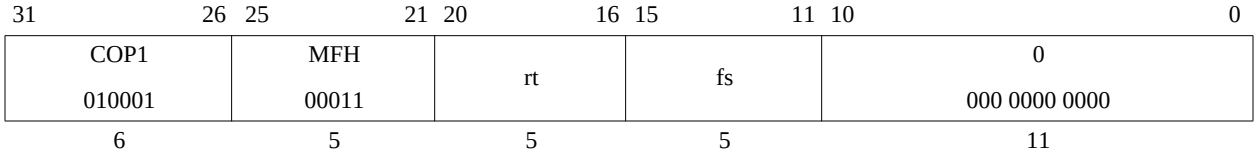
Operation:

```
data ← CP2CPR[Impl]31..0
GPR[rt] ← sign_extend(data)
```

Exceptions:

Coprocessor Unusable

Move Word From High Half of Floating Point Register MFHC1



Format: `MFHC1 rt, fs` **MIPS32 Release 2**

Purpose:

To copy a word from the high half of an FPU (CP1) general register to a GPR

Description: `GPR[rt] ← sign_extend(FPR[fs]`$_{63..32}$`)`

The contents of the high word of FPR *fs* are sign-extended and loaded into general register *rt*. This instruction is primarily intended to support 64-bit floating point units on a 32-bit CPU, but the semantics of the instruction are defined for all cases.

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

The results are **UNPREDICTABLE** if $Status_{FR} = 0$ and *fs* is odd.

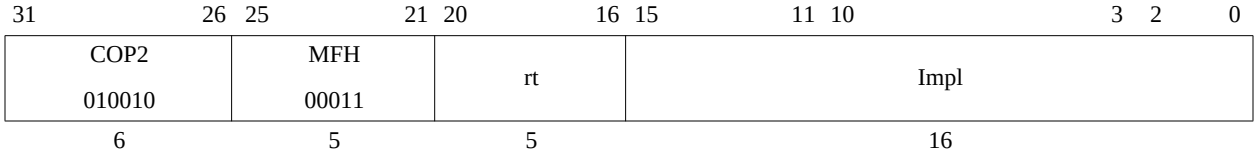
Operation:

```
data ← ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)63..32
GPR[rt] ← sign_extend(data)
```

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: `MFHC2 rt, rd` **MIPS32 Release 2**

`MFHC2, rt, rd, sel` **MIPS32 Release 2**

The syntax shown above is an example using MFHC1 as a model. The specific syntax is implementation dependent.

Purpose:

To copy a word from the high half of a COP2 general register to a GPR

Description: $GPR[rt] \leftarrow sign\_extend(CP2CPR[Impl]_{63..32})$

The contents of the high word of the coprocessor 2 register denoted by the *Impl* field are sign-extended and placed into GPR rt. The interpretation of the *Impl* field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:

The results are **UNPREDICTABLE** if *Impl* specifies a coprocessor 2 register that does not exist, or if that register is not 64 bits wide.

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

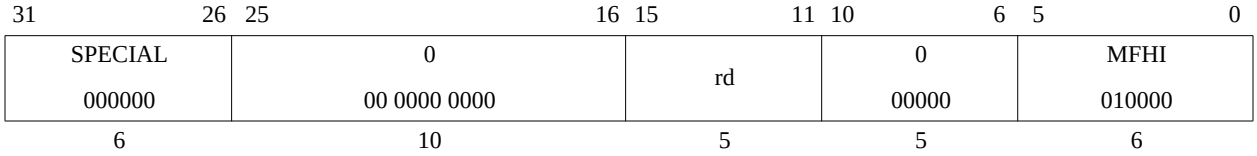
```
data ← CP2CPR[Impl]63..32
GPR[rt] ← sign_extend(data)
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Move From HI Register MFHI



Format: `MFHI rd` **MIPS32**

Purpose:

To copy the special purpose *HI* register to a GPR

Description: `GPR[rd] ← HI`

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

Restrictions:

None

Operation:

```
GPR[rd] ← HI
```

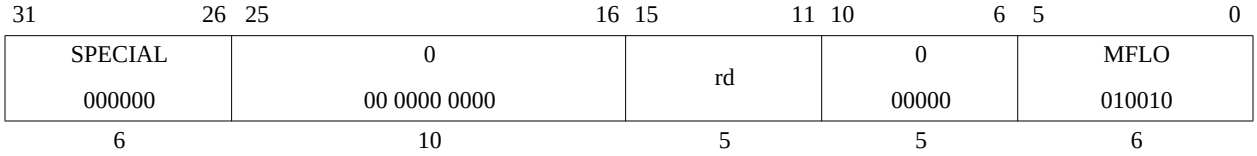
Exceptions:

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.

Move From LO Register MFLO



Format: `MFLO rd` **MIPS32**

Purpose:

To copy the special purpose *LO* register to a GPR

Description: `GPR[rd] ← LO`

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

Restrictions: None

Operation:

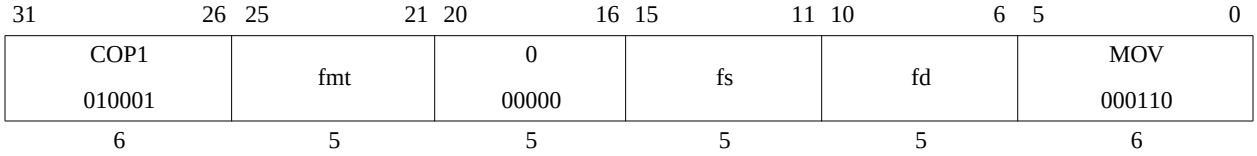
```
GPR[rd] ← LO
```

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.



Format: `MOV.S fd, fs` **MIPS32**

`MOV.D fd, fs` **MIPS32**

`MOV.PS fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

To move an FP value between FPRs

Description: `FPR[fd] ← FPR[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 **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

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

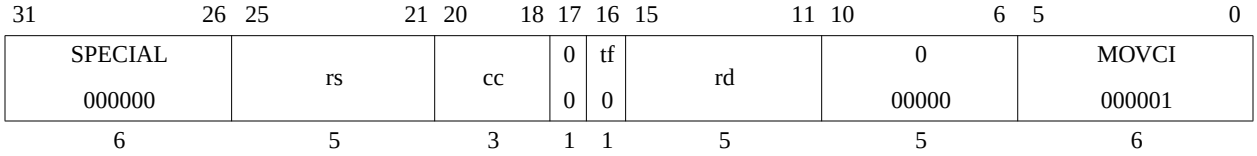
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Move Conditional on Floating Point False — MOVF



Format: `MOVF rd, rs, cc` **MIPS32**

Purpose:

To test an FP condition code then conditionally move a GPR

Description: `if FPConditionCode(cc) = 0 then GPR[rd] ← GPR[rs]`

If the floating point condition code specified by *CC* is zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

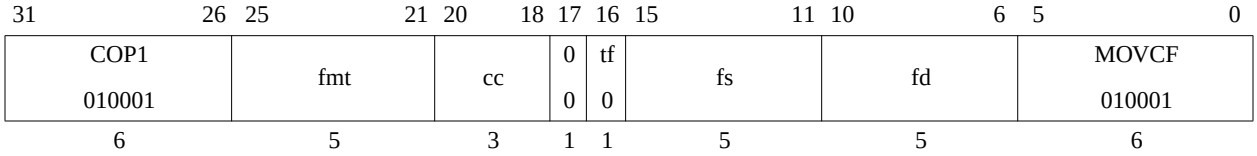
Operation:

```
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



Format: `MOVF.S fd, fs, cc` **MIPS32**

`MOVF.D fd, fs, cc` **MIPS32**

`MOVF.PS fd, fs, cc` **MIPS64**

MIPS32 Release 2

Purpose:

To test an FP condition code then conditionally move an FP value

Description: `if FPConditionCode(cc) = 0 then FPR[fd] ← FPR[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 **UNPREDICTABLE**. 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 **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
```

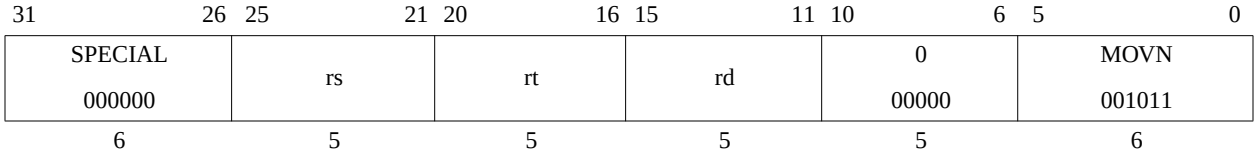
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions*:*

Unimplemented Operation

Move Conditional on Not Zero — MOVN



Format: `MOVN rd, rs, rt` **MIPS32**

Purpose:

To conditionally move a GPR after testing a GPR value

Description: `if GPR[rt] ≠ 0 then GPR[rd] ← GPR[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
```

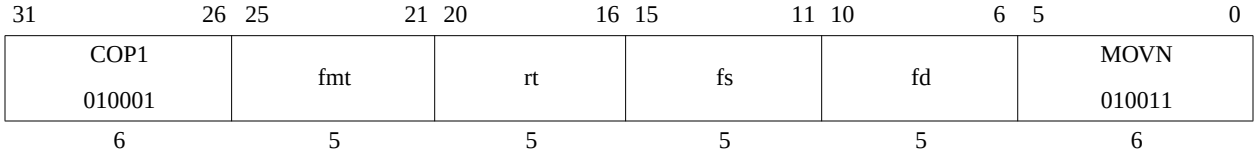
Exceptions:

None

Programming Notes:

The non-zero value tested here is the *condition true* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

Floating Point Move Conditional on Not Zero — MOVN.fmt



Format: `MOVN.S fd, fs, rt` **MIPS32**

`MOVN.D fd, fs, rt` **MIPS32**

`MOVN.PS fd, fs, rt` **MIPS64, MIPS32 Release 2**

Purpose:

To test a GPR then conditionally move an FP value

Description: `if GPR[rt] ≠ 0 then FPR[fd] ← FPR[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 **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

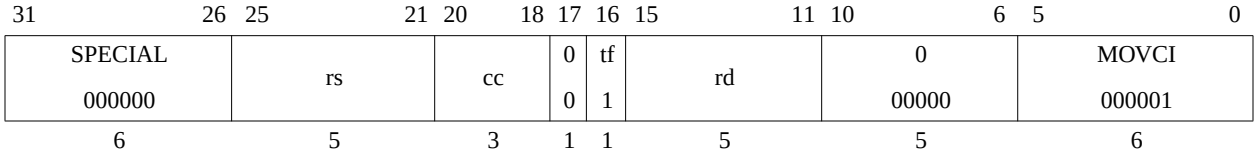
```
if GPR[rt] ≠ 0 then
   StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
   StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions*:*

Unimplemented Operation



Format: `MOVT rd, rs, cc` **MIPS32**

Purpose:

To test an FP condition code then conditionally move a GPR

Description: `if FPConditionCode(cc) = 1 then GPR[rd] ← GPR[rs]`

If the floating point condition code specified by *CC* is one, then the contents of GPR *rs* are placed into GPR *rd*.

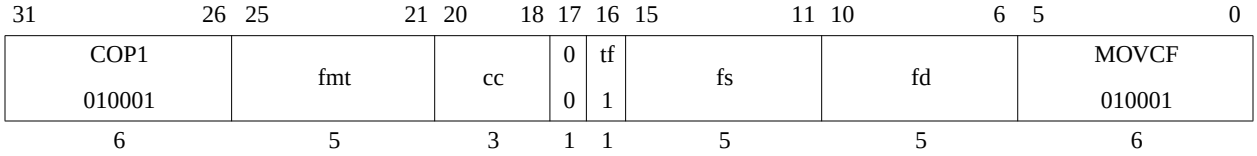
Restrictions:

Operation:

```
if FPConditionCode(cc) = 1 then
   GPR[rd] ← GPR[rs]
endif
```

Exceptions:

Reserved Instruction, Coprocessor Unusable



Format: `MOVT.S fd, fs, cc` **MIPS32**

`MOVT.D fd, fs, cc` **MIPS32**

`MOVT.PS fd, fs, cc` **MIPS64, MIPS32 Release 2**

Purpose:

To test an FP condition code then conditionally move an FP value

Description: `if FPConditionCode(cc) = 1 then FPR[fd] ← FPR[fs]`

If the floating point condition code specified by *CC* is one, then the value in FPR *fs* is placed into FPR *fd*. The source and destination are values in format *fmt*.

If the condition code is not one, then FPR *fs* is not copied and FPR *fd* contains its previous value in format *fm*t. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes undefined.

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 **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 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
```

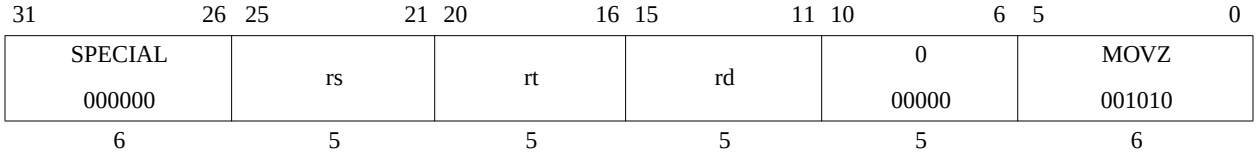
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Move Conditional on Zero MOVZ



Format: `MOVZ rd, rs, rt` **MIPS32**

Purpose:

To conditionally move a GPR after testing a GPR value

Description: `if GPR[rt] = 0 then GPR[rd] ← GPR[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:

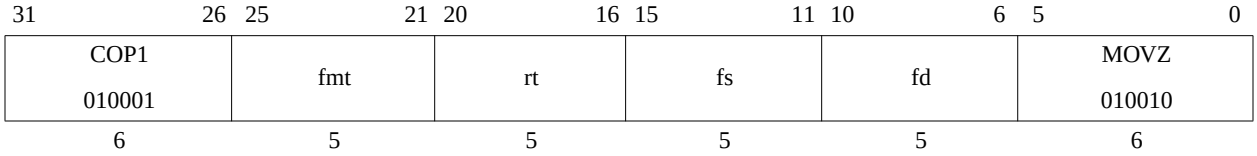
```
if GPR[rt] = 0 then
   GPR[rd] ← GPR[rs]
endif
```

Exceptions:

None

Programming Notes:

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



Format: `MOVZ.S fd, fs, rt` **MIPS32**

`MOVZ.D fd, fs, rt` **MIPS32**

`MOVZ.PS fd, fs, rt` **MIPS64, MIPS32 Release 2**

Purpose:

To test a GPR then conditionally move an FP value

Description: `if GPR[rt] = 0 then FPR[fd] ← FPR[fs]`

If the value in GPR *rt* is equal to zero then the value in FPR *fs* is placed in FPR *fd*. The source and destination are values in format *fmt*.

If GPR *rt* is not zero, then FPR *fs* is not copied and FPR *fd* contains its previous value in format *fm*t. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes **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 **UNPREDICTABLE** 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
```

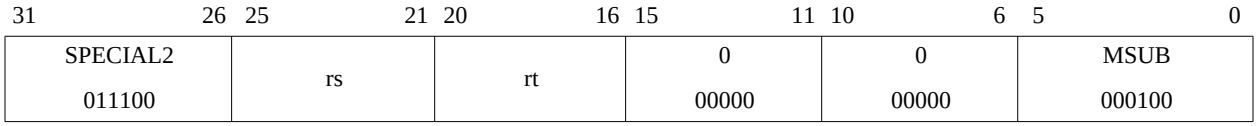
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Multiply and Subtract Word to Hi,Lo — MSUB



Format: `MSUB rs, rt` **MIPS32**

Purpose:

To multiply two words and subtract the result from Hi, Lo

Description: `(HI,LO) ← (HI,LO) - (GPR[rs] × GPR[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 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:

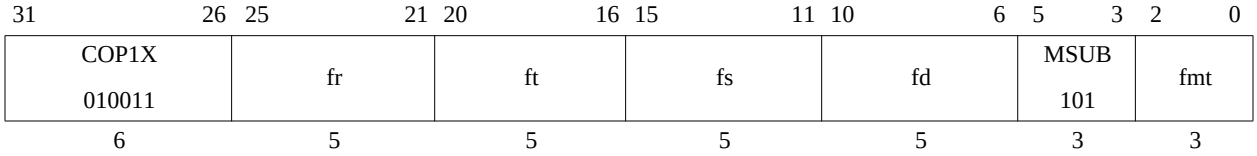
```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← (HI31..0 || LO31..0) - (GPR[rs]31..0 × GPR[rt]31..0)
HI ← sign_extend(temp63..32)
LO ← sign_extend(temp31..0)
```

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.



Format: `MSUB.S fd, fr, fs, ft` **MIPS64**

`MSUB.D fd, fr, fs, ft` **MIPS64**

`MSUB.PS fd, fr, fs, ft` **MIPS64, MIPS32 Release 2**

Purpose:

To perform a combined multiply-then-subtract of FP values

Description: `FPR[fd] ← (FPR[fs] × FPR[ft]) − FPR[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 **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 MSUB.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

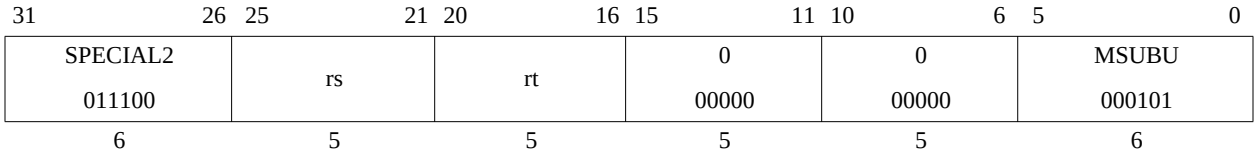
```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, (vfs ×fmt vft) −fmt vfr))
```

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: `(HI,LO) ← (HI,LO) - (GPR[rs] × GPR[rt])`

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)
```

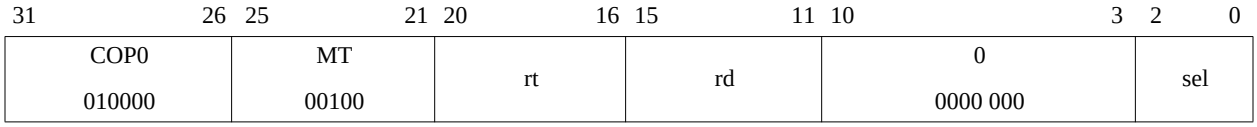
Exceptions:

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.

Move to Coprocessor 0 — MTC0



Format: `MTC0 rt, rd` **MIPS32**

`MTC0 rt, rd, sel` **MIPS32**

Purpose:

To move the contents of a general register to a coprocessor 0 register.

Description: `CPR[0, rd, sel] ← GPR[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:

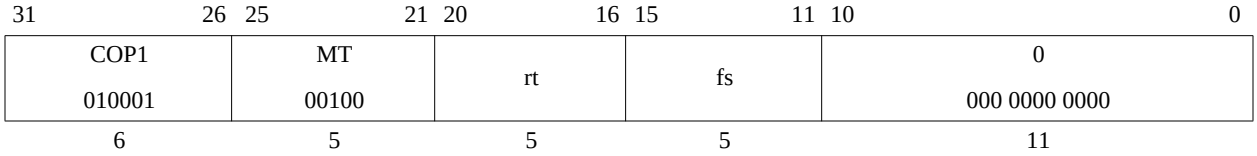
```
data ← GPR[rt]
if (Width(CPR[0,rd,sel]) = 64) then
   CPR[0,rd,sel] ← data
else
   CPR[0,rd,sel] ← data31..0
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Move Word to Floating Point — MTC1



Format: `MTC1 rt, fs` **MIPS32**

Purpose:

To copy a word from a GPR to an FPU (CP1) general register

Description: `FPR[fs] ← GPR[rt]`

The low word in GPR *rt* is placed into the low word of FPR *fs*. If FPRs are 64 bits wide, bits 63..32 of FPR *fs* become undefined.

Restrictions:

Operation:

```
data ← GPR[rt]31..0
StoreFPR(fs, UNINTERPRETED_WORD, data)
```

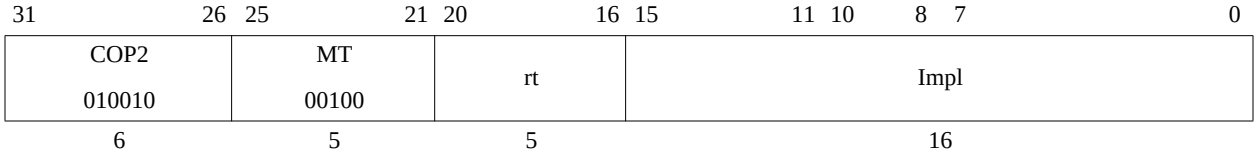
Exceptions:

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.

Move Word to Coprocessor 2 MTC2



Format: `MTC2 rt, rd` **MIPS32**

`MTC2 rt, rd, sel` **MIPS32**

The syntax shown above is an example using MTC1 as a model. The specific syntax is implementation dependent.

Purpose:

To copy a word from a GPR to a COP2 general register

Description: `CP2CPR[Impl] ← GPR[rt]`

The low word in GPR *rt* is placed into the low word of coprocessor 2 general register denoted by the *Impl* field. If coprocessor 2 general registers are 64 bits wide, bits 63..32 of the register denoted by the *Impl* field become undefined. The interpretation of the *Impl* field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:

The results are **UNPREDICTABLE** if *Impl* specifies a coprocessor 2 register that does not exist.

Operation:

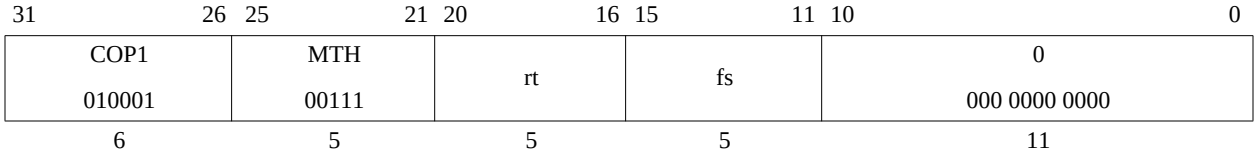
```
data ← GPR[rt]31..0
CP2CPR[Impl] ← data
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Move Word to High Half of Floating Point Register — MTHC1



Format: `MTHC1 rt, fs` **MIPS32 Release 2**

Purpose:

To copy a word from a GPR to the high half of an FPU (CP1) general register

Description: $FPR[fs]_{63..32} \leftarrow GPR[rt]_{31..0}$

The low word in GPR *rt* is placed into the high word of FPR *fs*. This instruction is primarily intended to support 64-bit floating point units on a 32-bit CPU, but the semantics of the instruction are defined for all cases.

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

The results are **UNPREDICTABLE** if $Status_{FR} = 0$ and *fs* is odd.

Operation:

```
newdata ← GPR[rt]31..0
       olddata ← ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)31..0
StoreFPR(fs, UNINTERPRETED_DOUBLEWORD, newdata || olddata)
```

Exceptions:

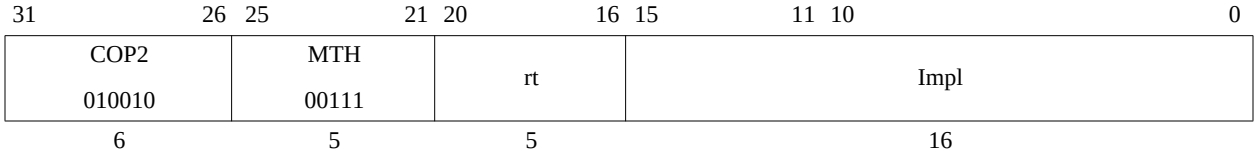
Coprocessor Unusable

Reserved Instruction

Programming Notes

When paired with MTC1 to write a value to a 64-bit FPR, the MTC1 must be executed first, followed by the MTHC1. This is because of the semantic definition of MTC1, which is not aware that software will be using an MTHC1 instruction to complete the operation, and sets the upper half of the 64-bit FPR to an **UNPREDICTABLE** value.

Move Word to High Half of Coprocessor 2 Register MTHC2



Format: `MTHC2 rt, rd` **MIPS32 Release 2**

`MTHC2 rt, rd, sel` **MIPS32 Release 2**

The syntax shown above is an example using MTHC1 as a model. The specific syntax is implementation dependent.

Purpose:

To copy a word from a GPR to the high half of a COP2 general register

Description: $\texttt{CP2CPR[Impl]}_{63..32} \leftarrow \texttt{GPR[rt]}_{31..0}$

The low word in GPR *rt* is placed into the high word of coprocessor 2 general register denoted by the *Impl* field. The interpretation of the *Impl* field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:

The results are **UNPREDICTABLE** if *Impl* specifies a coprocessor 2 register that does not exist, or if that register is not 64 bits wide.

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

```
data ← GPR[rt]31..0
CP2CPR[Impl] ← data || CPR[2,rd,sel]31..0
```

Exceptions:

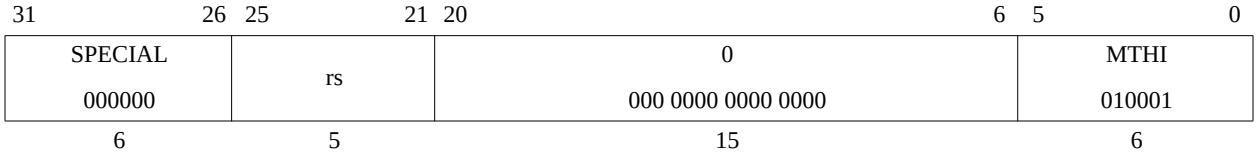
Coprocessor Unusable

Reserved Instruction

Programming Notes

When paired with MTC2 to write a value to a 64-bit CPR, the MTC2 must be executed first, followed by the MTHC2. This is because of the semantic definition of MTC2, which is not aware that software will be using an MTHC2 instruction to complete the operation, and sets the upper half of the 64-bit CPR to an **UNPREDICTABLE** value.

Move to HI Register MTHI



Format: `MTHI rs` **MIPS32**

Purpose:

To copy a GPR to the special purpose *HI* register

Description: `HI ← GPR[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]
```

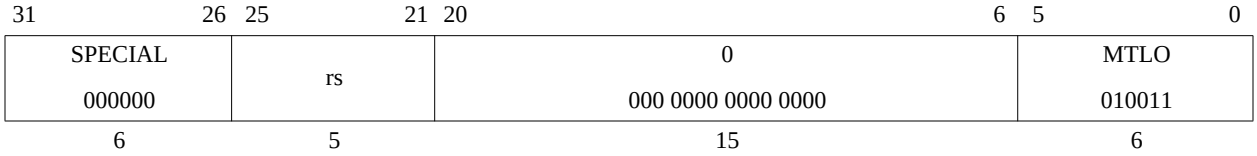
Exceptions:

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.

Move to LO Register MTLO



Format: `MTLO rs` **MIPS32**

Purpose:

To copy a GPR to the special purpose *LO* register

Description: `LO ← GPR[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]
```

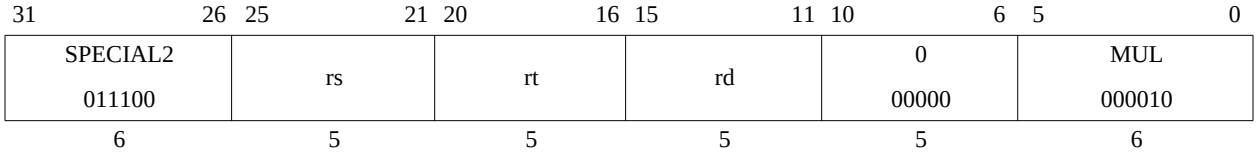
Exceptions:

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



Format: `MUL rd, rs, rt` **MIPS32**

Purpose:

To multiply two words and write the result to a GPR.

Description: `GPR[rd] ← GPR[rs] × GPR[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
   UNPREDICTABLE
endif
temp <- GPR[rs] * GPR[rt]
GPR[rd] <- sign_extend(temp31..0)
HI <- UNPREDICTABLE
LO <- UNPREDICTABLE
```

Exceptions:

None

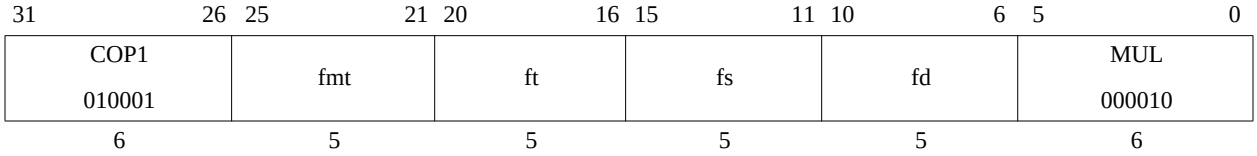
Programming Notes:

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 *GPR rd* 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 Multiply MUL.fmt



Format: `MUL.S fd, fs, ft` **MIPS32**

`MUL.D fd, fs, ft` **MIPS32**

`MUL.PS fd, fs, ft` **MIPS64**

MIPS32 Release 2

Purpose:

To multiply FP values

Description: `FPR[fd] ← FPR[fs] × FPR[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:

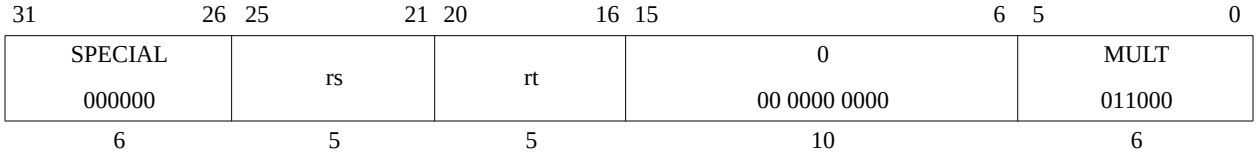
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

Multiply Word

MULT



Format: `MULT rs, rt` **MIPS32**

Purpose:

To multiply 32-bit signed integers

Description: `(HI, LO) ← GPR[rs] ×GPR[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
   UNPREDICTABLE
endif
   prod   ← GPR[rs]31..0 × GPR[rt]31..0
   LO     ← sign_extend(prod31..0)
   HI     ← sign_extend(prod63..32)
```

Exceptions:

None

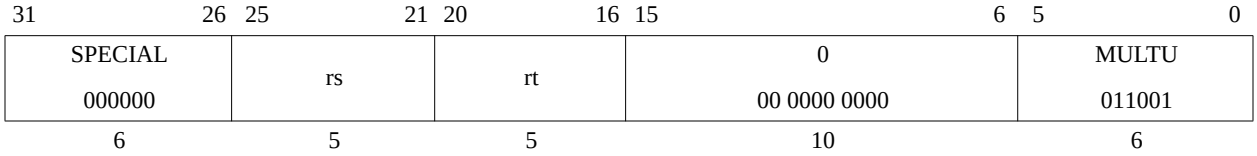
Programming Notes:

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.

Multiply Unsigned Word — MULTU



Format: `MULTU rs, rt` **MIPS32**

Purpose:

To multiply 32-bit unsigned integers

Description: `(HI, LO) ← GPR[rs] × GPR[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
   UNPREDICTABLE
endif
   prod← (0 || GPR[rs]31..0) × (0 || GPR[rt]31..0)
   LO ← sign_extend(prod31..0)
   HI ← sign_extend(prod63..32)
```

Exceptions:

None

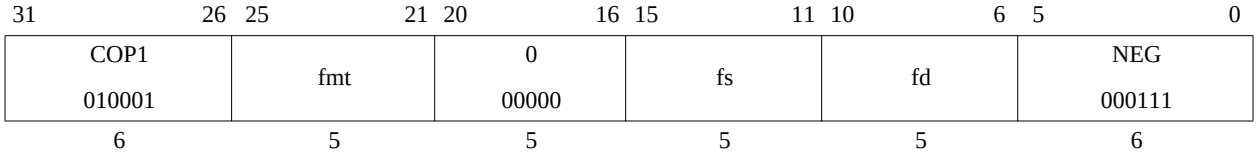
Programming Notes:

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



Format: `NEG.S fd, fs` **MIPS32**

`NEG.D fd, fs` **MIPS32**

`NEG.PS fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

To negate an FP value

Description: `FPR[fd] ← −FPR[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:

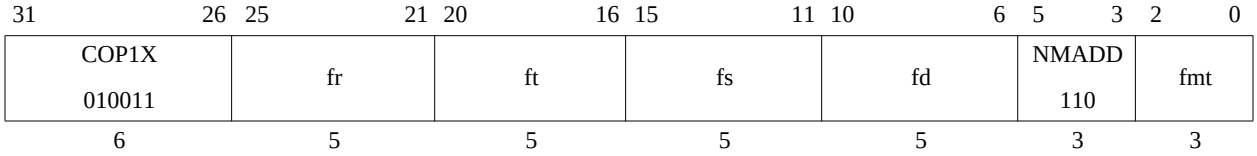
```
StoreFPR(fd, fmt, Negate(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation



Format: `NMADD.S fd, fr, fs, ft` **MIPS64**

`NMADD.D fd, fr, fs, ft` **MIPS64**

`NMADD.PS fd, fr, fs, ft` **MIPS64, MIPS32 Release 2**

Purpose:

To negate a combined multiply-then-add of FP values

Description: `FPR[fd] ← − ((FPR[fs] × FPR[ft]) + FPR[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 **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 NMADD.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

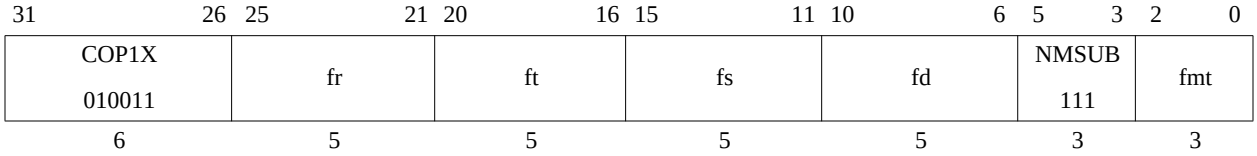
```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, −(vfr +fmt (vfs ×fmt vft)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow



Format: `NMSUB.S fd, fr, fs, ft` **MIPS64**

`NMSUB.D fd, fr, fs, ft` **MIPS64**

`NMSUB.PS fd, fr, fs, ft` **MIPS64, MIPS32 Release 2**

Purpose:

To negate a combined multiply-then-subtract of FP values

Description: `FPR[fd] ← - ((FPR[fs] × FPR[ft]) - FPR[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:

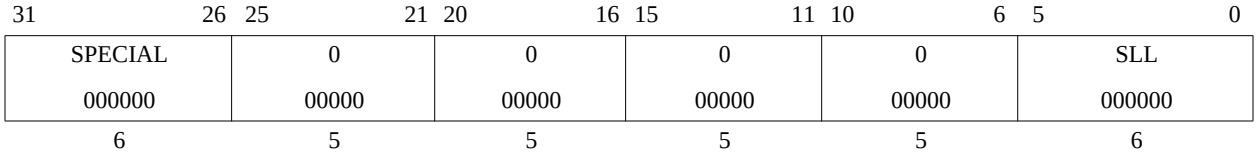
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

No Operation

NOP



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:

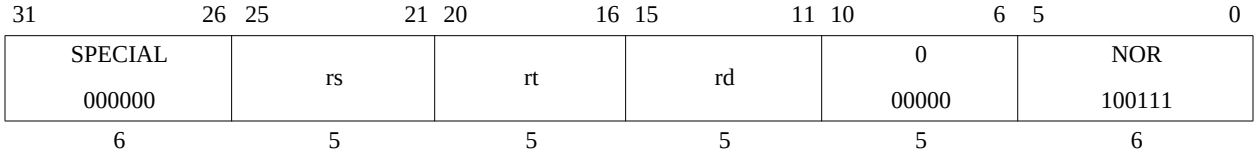
None

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.

Not Or

NOR



Format: `NOR rd, rs, rt` **MIPS32**

Purpose:

To do a bitwise logical NOT OR

Description: `GPR[rd] ← GPR[rs] NOR GPR[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:

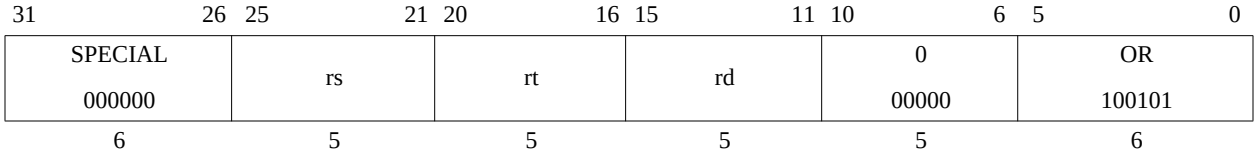
None

Operation:

```
GPR[rd] ← GPR[rs] nor GPR[rt]
```

Exceptions:

None



Format: `OR rd, rs, rt` **MIPS32**

Purpose:

To do a bitwise logical OR

Description: `GPR[rd] ← GPR[rs] or GPR[rt]`

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

Restrictions:

None

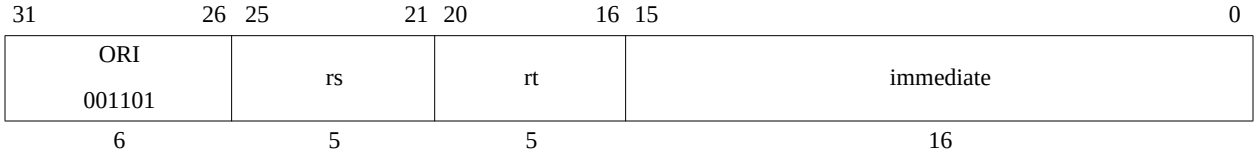
Operation:

```
GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:

None

Or Immediate ORI



Format: `ORI rt, rs, immediate` **MIPS32**

Purpose:

To do a bitwise logical OR with a constant

Description: `GPR[rt] ← GPR[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:

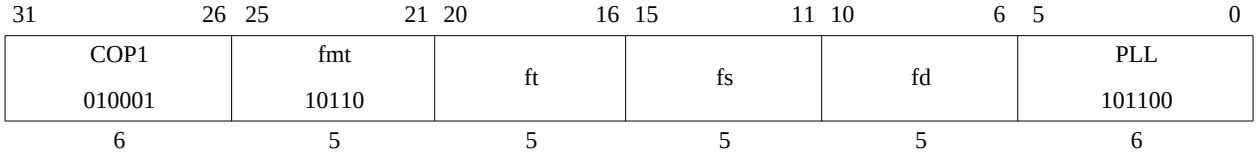
None

Operation:

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

Exceptions:

None



Format: `PLL.PS fd, fs, ft` **MIPS64, MIPS32 Release 2**

Purpose:

To merge a pair of paired single values with realignment

Description: `FPR[fd] ← lower(FPR[fs]) || lower(FPR[ft])`

A new paired-single value is formed by catenating the lower single of FPR *fs* (bits 31..0) and the lower single ofFPR *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 **UNPREDICTABLE**.

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

Operation:

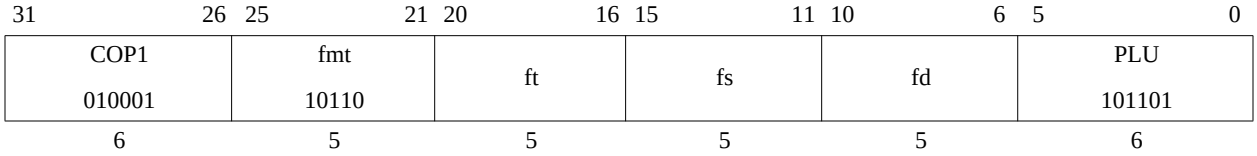
```
StoreFPR(fd, PS, ValueFPR(fs, PS)31..0 || ValueFPR(ft, PS)31..0)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Pair Lower Upper

PLU.PS



Format: `PLU.PS fd, fs, ft` **MIPS64, MIPS32 Release 2**

Purpose:

To merge a pair of paired single values with realignment

Description: `FPR[fd] ← lower(FPR[fs]) || upper(FPR[ft])`

A new paired-single value is formed by catenating the lower single of FPR *fs* (bits 31..0) and the upper single of FPR *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**.

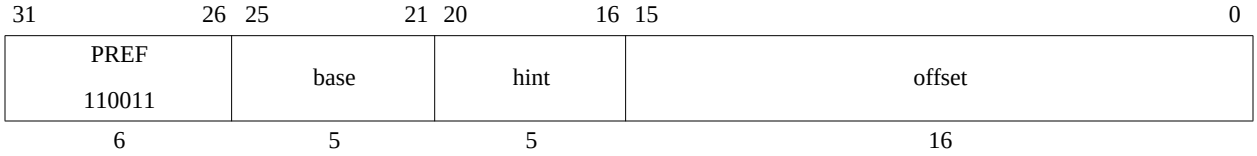
The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
StoreFPR(fd, PS, ValueFPR(fs, PS)31..0 || ValueFPR(ft, PS)63..32)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction



Format: `PREF hint,offset(base)` **MIPS32**

Purpose:

To move data between memory and cache.

Description: `prefetch_memory(GPR[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 causing data to be moved to or from the 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. The PrepareForStore function is unique in that it may modify the architecturally visible state.

PREF does not cause addressing-related exceptions, including TLB 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 moved, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREF instruction.

PREF neither generates a memory operation nor modifies the state of a cache line for a location with an *uncached* memory access type, whether this type is specified by the address segment (e.g., kseg1), the programmed coherency attribute of a segment (e.g., the use of the K0, KU, or K23 fields in the *Config* register), or the per-page coherency attribute provided by the TLB.

If PREF results in a memory operation, the memory access type and coherency attribute used for the operation are determined by the memory access type and coherency attribute 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.

Table 3-30 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). Action: Fetch data as if for a load.
1	store	Use: Prefetched data is expected to be stored or modified. 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. 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. 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. 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. 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 3-30 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. 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. If the cache line is not dirty, it is implementation dependent whether the state of the cache line is marked invalid or left unchanged. If the cache line is locked, no action is taken.
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. 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. Programming Note: Because the cache line is filled with zero data on a cache miss, software must not assume that this action, in and of itself, can be used as a fast bzero-type function.
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:

Bus Error, Cache Error

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:

Prefetch cannot move data to or 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.

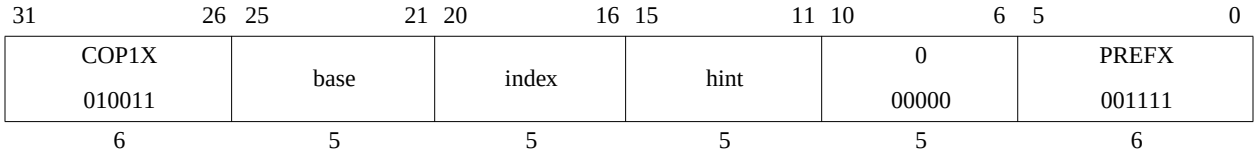
Prefetch does not cause addressing exceptions. A prefetch may be used using an address pointer before the validity of the pointer is determined without worrying about an addressing exception.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREF instruction. Typically, this only occurs in systems which have high-reliability requirements.

Prefetch operations have no effect on cache lines that were previously locked with the CACHE instruction.

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 — PREFX



Format: `PREFX hint, index(base)` **MIPS64**
MIPS32 Release 2

Purpose:

To move data between memory and cache.

Description: `prefetch_memory[GPR[base] + GPR[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:

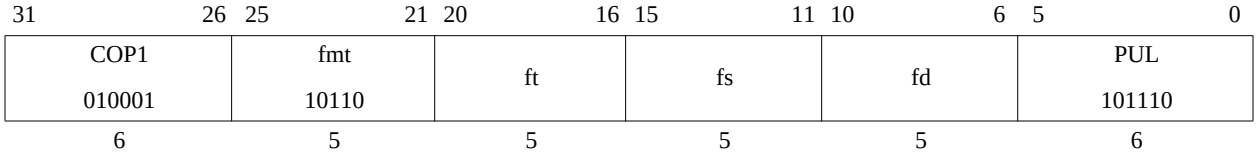
Coprocessor Unusable, Reserved Instruction, Bus Error, Cache Error

Programming Notes:

The PREFX instruction is only available on processors that implement floating point and should never by generated by compilers in situations other than those 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



Format: `PUL.PS fd, fs, ft` **MIPS64, MIPS32 Release 2**

Purpose:

To merge a pair of paired single values with realignment

Description: `FPR[fd] ← upper(FPR[fs]) || lower(FPR[ft])`

A new paired-single value is formed by catenating the upper single of FPR *fs* (bits 63..32) and the lower single of FPR *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 **UNPREDICTABLE**.

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

Operation:

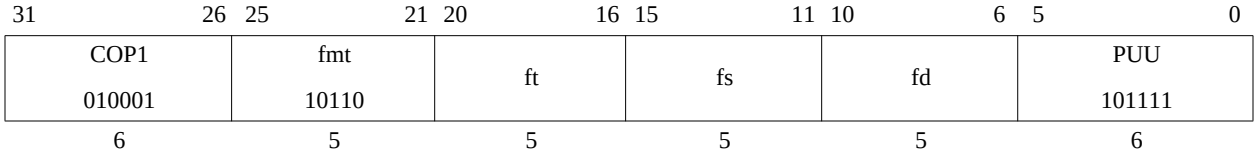
$$\texttt{StoreFPR(fd, PS, ValueFPR(fs, PS)}_{63..32}\ \texttt{||}\ \texttt{ValueFPR(ft, PS)}_{31..0}\texttt{)}$$

Exceptions:

Coprocessor Unusable, Reserved Instruction

Pair Upper Upper

PUU.PS



Format: `PUU.PS fd, fs, ft` **MIPS64, MIPS32 Release 2**

Purpose:

To merge a pair of paired single values with realignment

Description: `FPR[fd] ← upper(FPR[fs]) || upper(FPR[ft])`

A new paired-single value is formed by catenating the upper single of FPR *fs* (bits 63..32) and the upper single of FPR *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**.

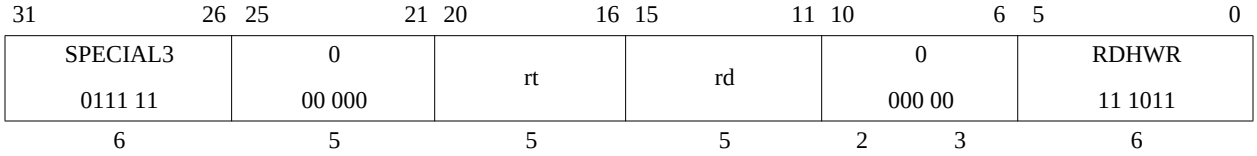
The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
StoreFPR(fd, PS, ValueFPR(fs, PS)63..32 || ValueFPR(ft, PS)63..32)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction



Format: `RDHWR rt,rd` **MIPS32 Release 2**

Purpose:

To move the contents of a hardware register to a general purpose register (GPR) if that operation is enabled by privileged software.

Description: `GPR[rt] ← HWR[rd]`

If access is allowed to the specified hardware register, the contents of the register specified by *rd* is sign-extended and loaded into general register *rt*. Access control for each register is selected by the bits in the coprocessor 0 *HWREna* register.

The available hardware registers, and the encoding of the *rd* field for each, are shown in Table 3-31.

Table 3-31 Hardware Register List

Register Number (*rd* Value)	Register Name	Contents
0	CPUNum	Number of the CPU on which the program is currently running. This comes directly from the coprocessor 0 $EBase_{CPUNum}$ field.
1	SYNCI_Step	Address step size to be used with the SYNCI instruction. See that instruction's description for the use of this value.
2	CC	High-resolution cycle counter. This comes directly from the coprocessor 0 *Count* register.
3	CCRes	Resolution of the CC register. This value denotes the number of cycles between update of the register. For example: CCRes Value / Meaning 1 — CC register increments every CPU cycle 2 — CC register increments every second CPU cycle 3 — CC register increments every third CPU cycle etc.
4-28		Reserved for future architectural use. Access results in a Reserved Instruction Exception.
29		Reserved for future use by a MIPS ABI extension. Access results in a Reserved Instruction Exception
30-31		These registers are reserved for implementation-dependent use. If they are implemented, the corresponding bits in the *HWREna* register control access. If they are not implemented, access results in a Reserved Instruction Exception.

Restrictions:

In implementations of Release 1 of the Architecture, this instruction resulted in a Reserved Instruction Exception.

Access to the specified hardware register is enabled if Coprocessor 0 is enabled, or if the corresponding bit is set in the *HWREna* register. If access is not allowed, a Reserved Instruction Exception is signaled.

Operation:

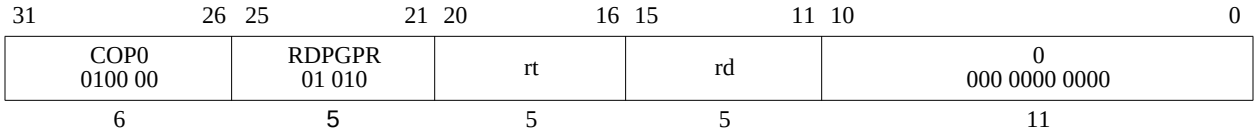
```
case rd
       0x00:  temp ← sign_extend(EBase_CPUNum)
   0x01:  temp ← sign_extend(SYNCI_StepSize())
   0x02:  temp ← sign_extend(Count)
       0x03:  temp ← sign_extend(CountResolution())
       0x30:  temp ← sign_extend_if_32bit_op(Implementation-Dependent-Value)
       0x31:  temp ← sign_extend_if_32bit_op(Implementation-Dependent-Value)
   otherwise: SignalException(ReservedInstruction)
endcase
GPR[rt] ←temp

function sign_extend_if_32bit_op(value)
   if (width(value) = 64) and Are64bitOperationsEnabled() then
      sign_extend_if_32bit_op ←value
   else
      sign_extend_if_32bit_op ←sign_extend(value)
   endif
end sign_extend_if_32bit_op
```

Exceptions:

Reserved Instruction

Read GPR from Previous Shadow Set — RDPGPR



Format: `RDPGPR rd, rt` **MIPS32 Release 2**

Purpose:

To move the contents of a GPR from the previous shadow set to a current GPR.

Description: `GPR[rd] ← SGPR[SRSCtl`$_{PSS}$`, rt]`

The contents of the shadow GPR register specified by $SRSCtl_{PSS}$ (signifying the previous shadow set number) and *rt* (specifying the register number within that set) is moved to the current GPR *rd*.

Restrictions:

In implementations prior to Release 2 of the Architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

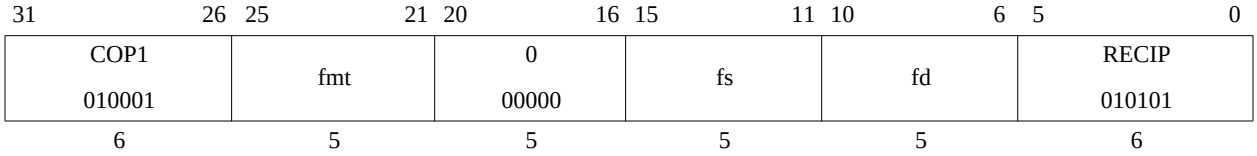
```
GPR[rd] ← SGPR[SRSCtl_PSS, rt]
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Reciprocal Approximation RECIP.fmt



Format: `RECIP.S fd, fs` **MIPS64, MIPS32 Release 2**

`RECIP.D fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

To approximate the reciprocal of an FP value (quickly)

Description: `FPR[fd] ← 1.0 / FPR[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 **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 RECIP.D is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
StoreFPR(fd, fmt, 1.0 / valueFPR(fs, fmt))
```

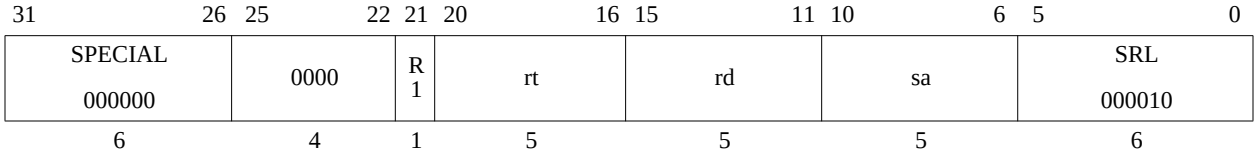
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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

Rotate Word Right — ROTR



Format: `ROTR rd, rt, sa` **SmartMIPS Crypto, MIPS32 Release 2**

Purpose:

To execute a logical right-rotate of a word by a fixed number of bits

Description: `GPR[rd] ← GPR[rt] ↔(right) sa`

The contents of the low-order 32-bit word of GPR *rt* are rotated right; the word result is sign-extended and placed in GPR *rd*. The bit-rotate amount is specified by *sa*.

Restrictions:

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:

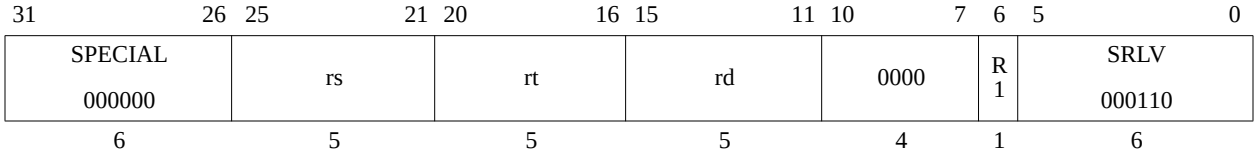
```
if NotWordValue(GPR[rt]) or
   ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
   UNPREDICTABLE
endif
s      ← sa
temp   ← GPR[rt]s-1..0 || GPR[rt]31..s
GPR[rd]← sign_extend(temp)
```

Exceptions:

Reserved Instruction

Rotate Word Right Variable

ROTRV



Format: `ROTRV rd, rt, rs` **SmartMIPS Crypto, MIPS32 Release 2**

Purpose:

To execute a logical right-rotate of a word by a variable number of bits

Description: `GPR[rd] ← GPR[rt] ↔(right) GPR[rs]`

The contents of the low-order 32-bit word of GPR *rt* are rotated right; the word result is sign-extended and placed in GPR *rd*. The bit-rotate amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

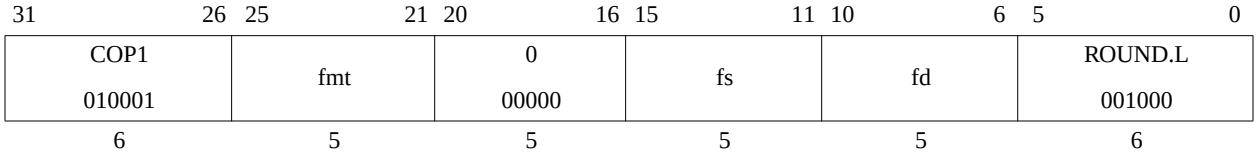
If GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if NotWordValue(GPR[rt]) or
   ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
   UNPREDICTABLE
endif
s      ← GPR[rs]4..0
temp   ← GPR[rt]s-1..0 || GPR[rt]31..s
GPR[rd]← sign_extend(temp)
```

Exceptions:

Reserved Instruction



Format: `ROUND.L.S fd, fs` **MIPS64, MIPS32 Release 2**
`ROUND.L.D fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

To convert an FP value to 64-bit fixed point, rounding to nearest

Description: `FPR[fd] ← convert_and_round(FPR[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))
```

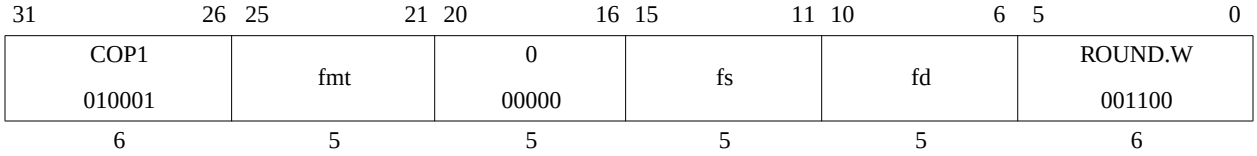
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow

Floating Point Round to Word Fixed Point — ROUND.W.fmt



Format: `ROUND.W.S fd, fs` **MIPS32**
`ROUND.W.D fd, fs` **MIPS32**

Purpose:

To convert an FP value to 32-bit fixed point, rounding to nearest

Description: `FPR[fd] ← convert_and_round(FPR[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))
```

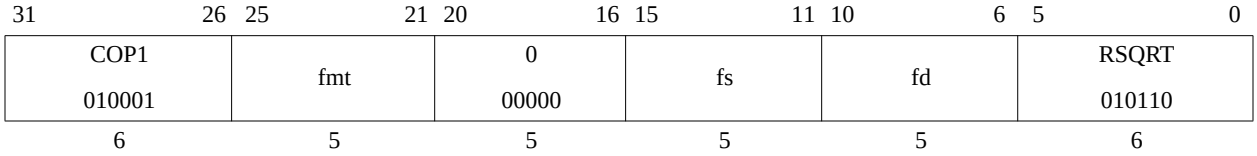
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow

Reciprocal Square Root Approximation RSQRT.fmt



Format: `RSQRT.S fd, fs` **MIPS64, MIPS32 Release 2**

`RSQRT.D fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

To approximate the reciprocal of the square root of an FP value (quickly)

Description: `FPR[fd] ← 1.0 / sqrt(FPR[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 **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 RSQRT.D is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
StoreFPR(fd, fmt, 1.0 / SquareRoot(valueFPR(fs, fmt)))
```

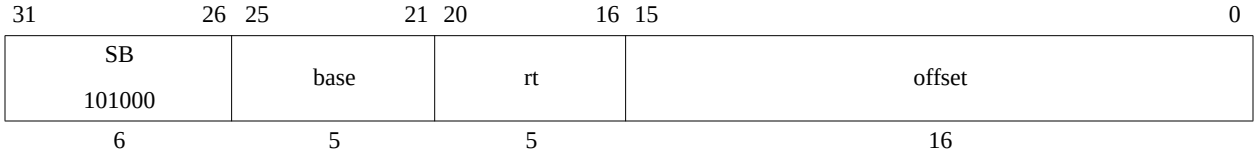
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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

Store Byte SB



Format: `SB rt, offset(base)` **MIPS32**

Purpose:

To store a byte to memory

Description: `memory[GPR[base] + offset] ← GPR[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

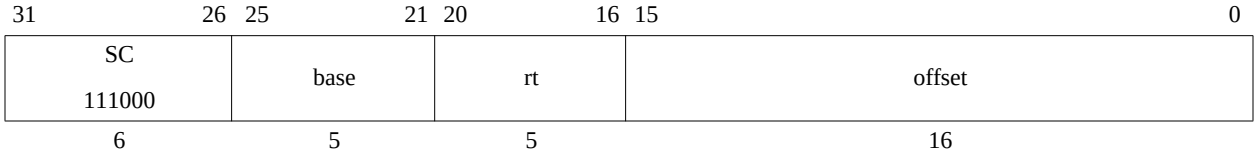
Operation:

```
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, Watch

Store Conditional Word SC



Format: `SC rt, offset(base)` **MIPS32**

Purpose:

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

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

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The least-significant 32-bit word in GPR *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 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 synchronizable 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 ERET instruction is executed.

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 memory access instruction (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 **UNPREDICTABLE**:

- Execution of SC must have been preceded by execution of an LL instruction.
- An RMW sequence executed without intervening events that would cause the SC to fail 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 synchronizable memory locations. A synchronizable memory location is one that is associated with the state and logic necessary to implement the LL/SC semantics. Whether a memory location is synchronizable depends on the processor and system configurations, and on the memory access type used for the location:

- **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.
- **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*.
- **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 **UNPREDICTABLE**.

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 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
bytesel← vAddr2..0 xor (BigEndianCPU || 0^2)
datadoubleword← GPR[rt]63-8*bytesel..0 || 0^8*bytesel
if LLbit then
   StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
endif
GPR[rt]← 0^63 || LLbit
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

Programming Notes:

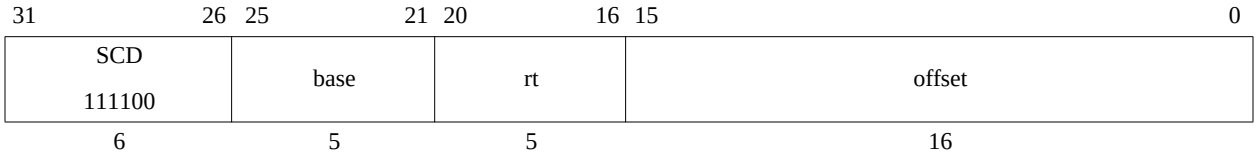
LL and SC are used to atomically update memory locations, as shown below.

```
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



Format: `SCD rt, offset(base)` **MIPS64**

Purpose:

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

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

The LLD and SCD instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The 64-bit doubleword in GPR *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 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:

- A coherent store is completed by another processor or coherent I/O module into the block of synchronizable physical memory containing the doubleword. 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 ERET instruction is executed.

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 **UNPREDICTABLE**:

- Execution of the SCD must be preceded by execution of an LLD instruction.
- An RMW sequence executed without intervening events that would cause the SCD to fail 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 synchronizable memory locations. A synchronizable memory location is one that is associated with the state and logic necessary to implement the LL/SC semantics. Whether a memory location is synchronizable depends on the processor and system configurations, and on the memory access type used for the location:

- **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.
- **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*.
- **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 **UNPREDICTABLE**.

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 ≠ 0^3 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] ← 0^63 || LLbit
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Reserved Instruction, Watch

Programming Notes:

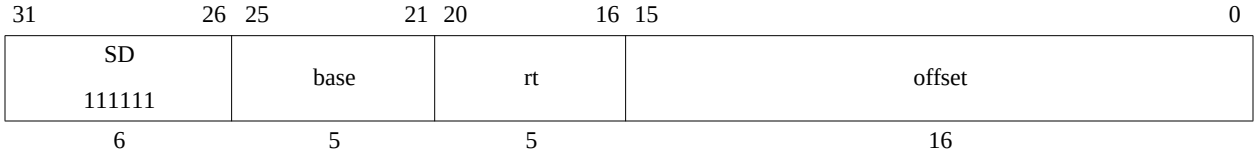
LLD and SCD are used to atomically update memory locations, as shown below.

```
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



Format: `SD rt, offset(base)` **MIPS64**

Purpose:

To store a doubleword to memory

Description: `memory[GPR[base] + offset] ← GPR[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:

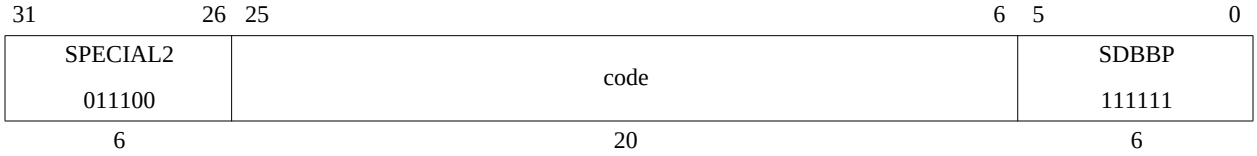
The effective address must be naturally-aligned. If any of the 3 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

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, Watch



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. If the processor is executing in Debug Mode when the SDBBP instruction is executedthe exception is a Debug Mode Exception, which sets the $Debug_{DExcCode}$ field to the value 0x9 (Bp). 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:

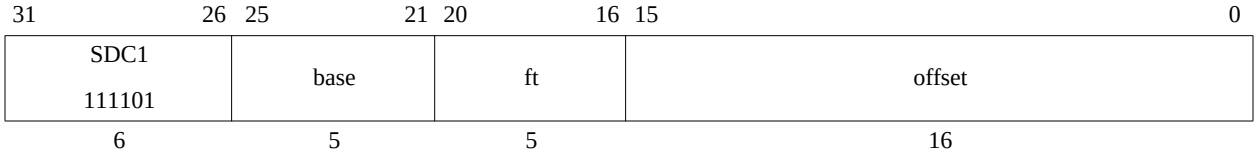
Operation:

```
If Debug_DM = 0 then
   SignalDebugBreakpointException()
else
   SignalDebugModeBreakpointException()
endif
```

Exceptions:

Debug Breakpoint Exception
Debug Mode Breakpoint Exception

Store Doubleword from Floating Point — SDC1



Format: `SDC1 ft, offset(base)` **MIPS32**

Purpose:

To store a doubleword from an FPR to memory

Description: `memory[GPR[base] + offset] ←FPR[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$_{2..0} \neq 0$ (not doubleword-aligned).

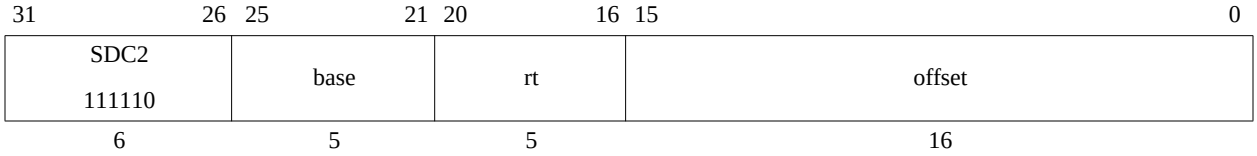
Operation:

```
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, Watch

Store Doubleword from Coprocessor 2 — SDC2



Format: `SDC2 rt, offset(base)` **MIPS32**

Purpose:

To store a doubleword from a Coprocessor 2 register to memory

Description: `memory[GPR[base] + offset] ← CPR[2,rt,0]`

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 $EffectiveAddress_{2..0} \neq 0$ (not doubleword-aligned).

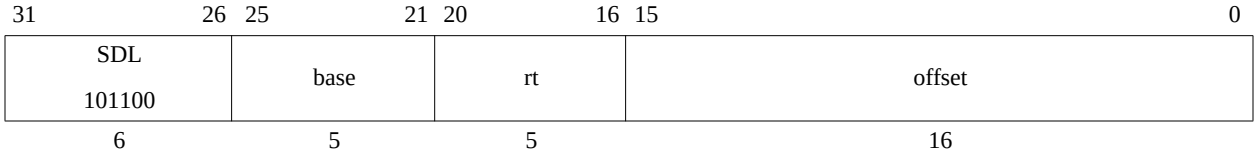
Operation:

```
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, Watch

Store Doubleword Left SDL



Format: `SDL rt, offset(base)` **MIPS64**

Purpose:

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

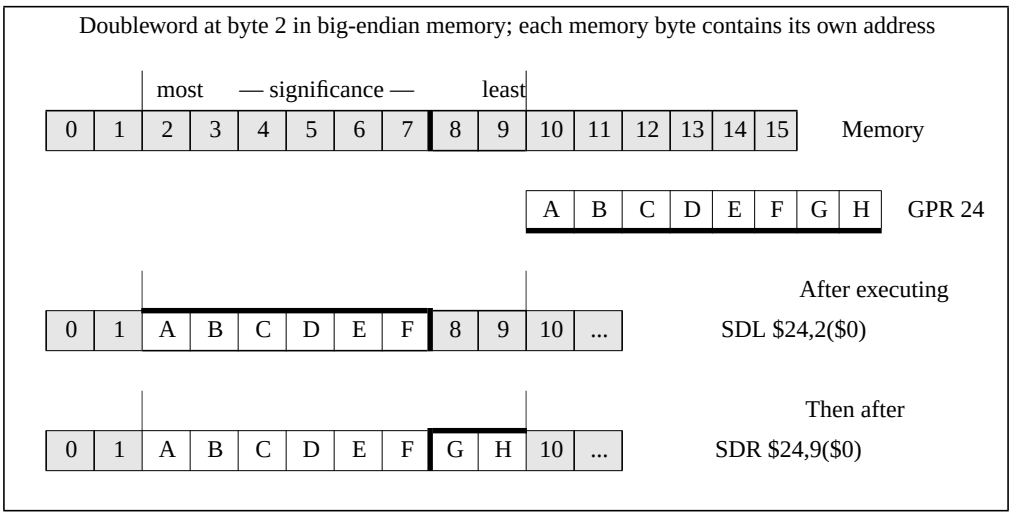
Description: `memory[GPR[base] + offset] ← Some_Bytes_From GPR[rt]`

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.*

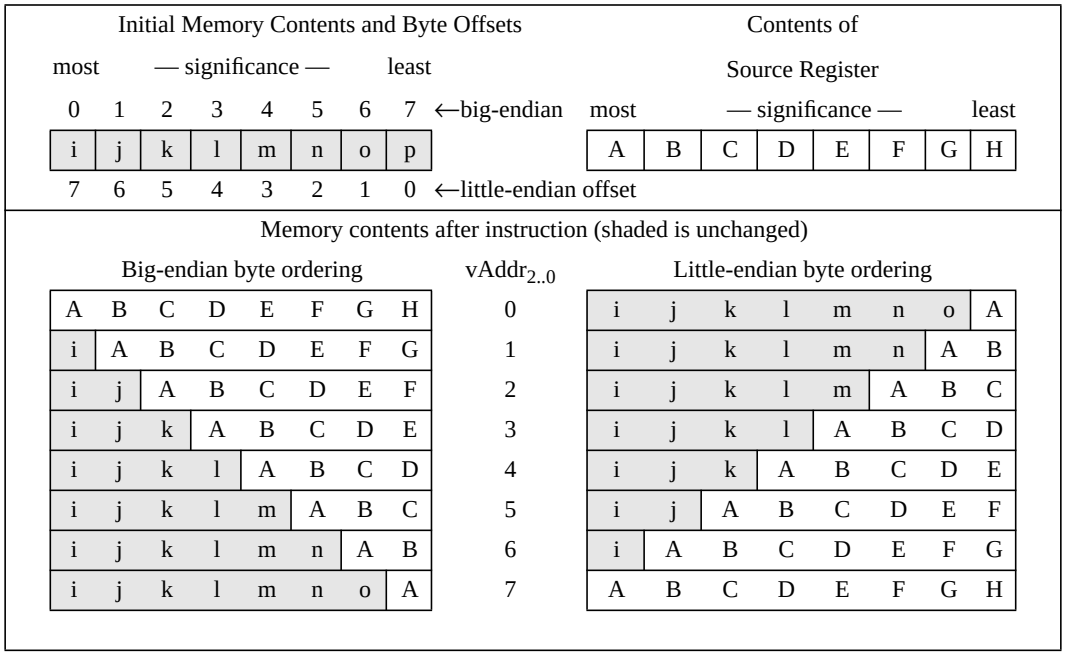
The figure below illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 6 bytes, is located in the aligned doubleword containing the most-significant byte at 2. First, SDL stores the 6 most-significant bytes of the source register into these bytes in memory. Next, the complementary SDR instruction stores the remainder of *DW.*

Figure 3-19 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 3-20 Bytes Stored by an SDL Instruction



Restrictions:

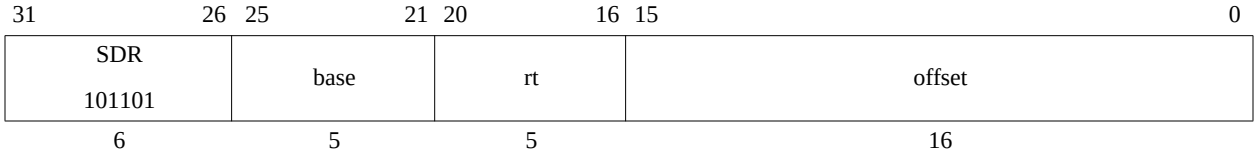
Operation:

```
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, Watch

Store Doubleword Right SDR



Format: `SDR rt, offset(base)` **MIPS64**

Purpose:

To store the least-significant part of a doubleword to an unaligned memory address

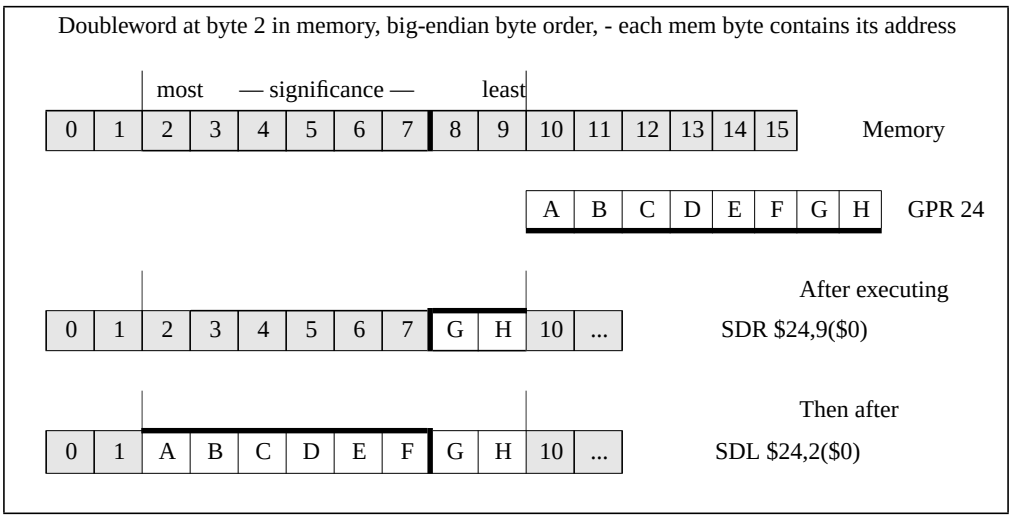
Description: `memory[GPR[base] + offset] ← Some_Bytes_From GPR[rt]`

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*.

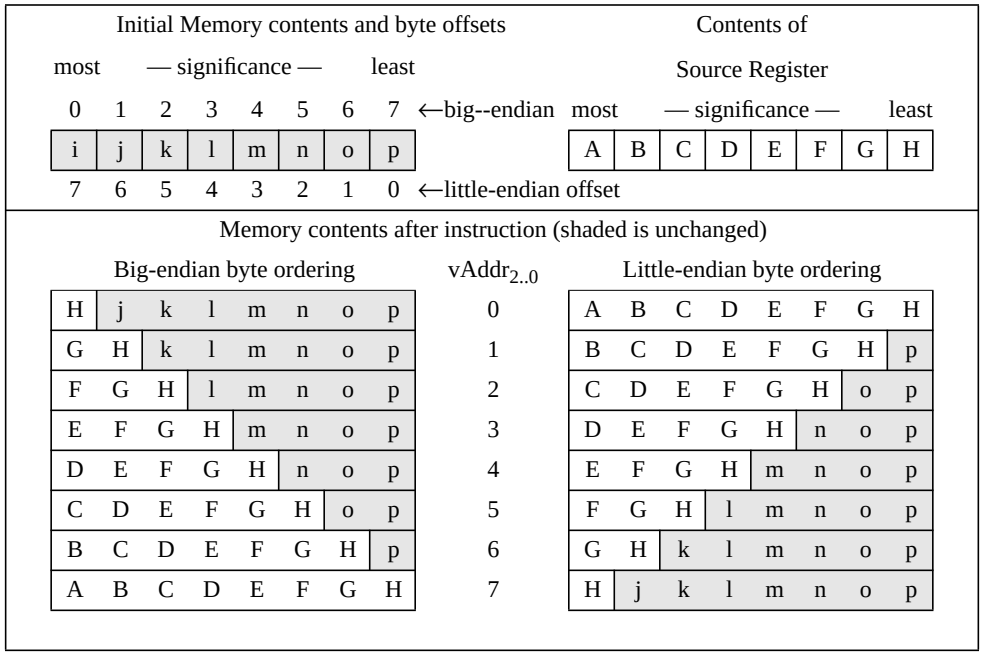
The figure below illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 2 bytes, is located in the aligned doubleword containing the least-significant byte at 9. First, SDR stores the 2 least-significant bytes of the source register into these bytes in memory. Next, the complementary SDL stores the remainder of *DW*.

Figure 3-21 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 3-22 shows the bytes stored for every combination of offset and byte-ordering.

Figure 3-22 Bytes Stored by an SDR Instruction



Restrictions:

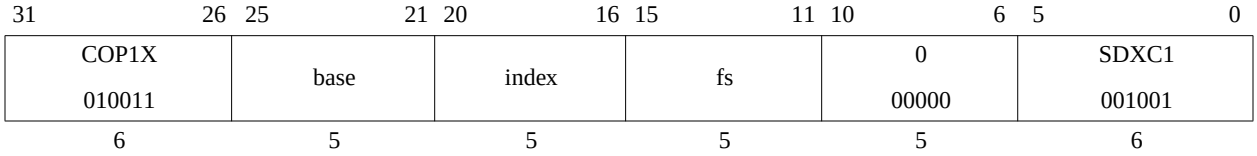
Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0  xor  ReverseEndian^3)
If BigEndianMem = 0 then
   pAddr← pAddr_PSIZE-1..3 || 0^3
endif
bytesel← vAddr_1..0 xor BigEndianCPU^3
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, Watch

Store Doubleword Indexed from Floating Point — SDXC1



Format: `SDXC1 fs, index(base)` **MIPS64**
MIPS32 Release 2

Purpose:

To store a doubleword from an FPR to memory (GPR+GPR addressing)

Description: `memory[GPR[base] + GPR[index]] ← FPR[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$_{2..0} \neq 0$ (not doubleword-aligned).

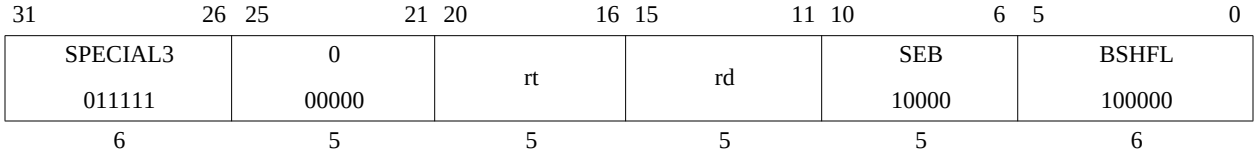
Operation:

```
vAddr ← GPR[base] + GPR[index]
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:

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

Sign-Extend Byte SEB



Format: `seb rd, rt` **MIPS32 Release 2**

Purpose:

To sign-extend the least significant byte of GPR *rt* and store the value into GPR *rd*.

Description: `GPR[rd] ← SignExtend(GPR[rt]`$_{7..0}$`)`

The least significant byte from GPR *rt* is sign-extended and stored in GPR *rd*.

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

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
   UNPREDICTABLE
endif
GPR[rd] ←sign_extend(GPR[rt]7..0)
```

Exceptions:

Reserved Instruction

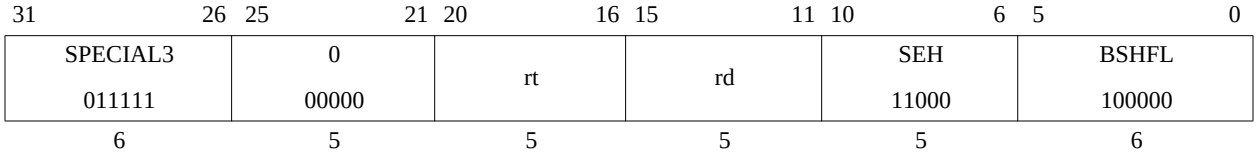
Programming Notes:

For symmetry with the SEB and SEH instructions, one would expect that there would be ZEB and ZEH instructions that zero-extend the source operand. Similarly, one would expect that the SEW and ZEW instructions would exist to sign- or zero-extend a word to a doubleword. These instructions do not exist because there are functionally-equivalent instructions already in the instruction set. The following table shows the instructions providing the equivalent functions.

Expected Instruction	Function	Equivalent Instruction
`ZEB rx,ry`	Zero-Extend Byte	`ANDI rx,ry,0xFF`
`ZEH rx,ry`	Zero-Extend Halfword	`ANDI rx,ry,0xFFFF`
`SEW rx,ry`	Sign-Extend Word	`SLL rx,ry,0`
`ZEW rx,rx`[1]	Zero-Extend Word	`DINSP32 rx,r0,32,32`

1. The equivalent instruction uses rx for both source and destination, so the expected instruction is limited to one register

Sign-Extend Halfword SEH



Format: `seh rd, rt` **MIPS32 Release 2**

Purpose:

To sign-extend the least significant halfword of GPR *rt* and store the value into GPR *rd*.

Description: $GPR[rd] \leftarrow SignExtend(GPR[rt]_{15..0})$

The least significant halfword from GPR *rt* is sign-extended and stored in GPR *rd*.

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

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
   UNPREDICTABLE
endif
GPR[rd] ←sign_extend(GPR[rt]15..0)
```

Exceptions:

Reserved Instruction

Programming Notes:

The SEH instruction can be used to convert two contiguous halfwords to sign-extended word values in three instructions. For example:

```
lw     t0, 0(a1)           /* Read two contiguous halfwords */
seh    t1, t0              /* t1 = lower halfword sign-extended to word */
sra    t0, t0, 16          /* t0 = upper halfword sign-extended to word */
```

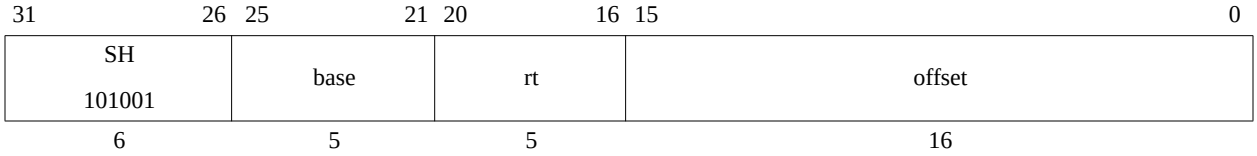
Zero-extended halfwords can be created by changing the SEH and SRA instructions to ANDI and SRL instructions, respectively.

For symmetry with the SEB and SEH instructions, one would expect that there would be ZEB and ZEH instructions that zero-extend the source operand. Similarly, one would expect that the SEW and ZEW instructions would exist to sign- or zero-extend a word to a doubleword. These instructions do not exist because there are functionally-equivalent instructions already in the instruction set. The following table shows the instructions providing the equivalent functions.

Expected Instruction	Function	Equivalent Instruction
`ZEB rx,ry`	Zero-Extend Byte	`ANDI rx,ry,0xFF`
`ZEH rx,ry`	Zero-Extend Halfword	`ANDI rx,ry,0xFFFF`
`SEW rx,ry`	Sign-Extend Word	`SLL rx,ry,0`
`ZEW rx,rx`[1]	Zero-Extend Word	`DINSP32 rx,r0,32,32`

1. The equivalent instruction uses rx for both source and destination, so the expected instruction is limited to one register

Store Halfword SH



Format: `SH rt, offset(base)` **MIPS32**

Purpose:

To store a halfword to memory

Description: `memory[GPR[base] + offset] ← GPR[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.

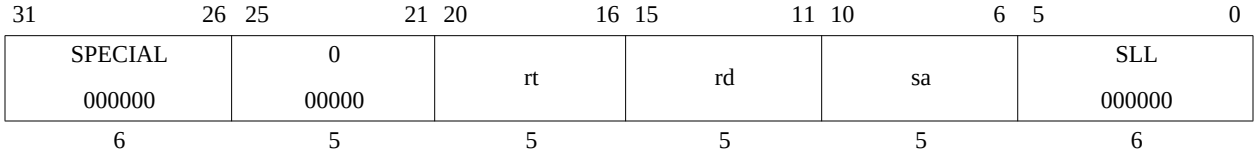
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr12..0 xor (ReverseEndian2 || 0))
bytesel← vAddr12..0 xor (BigEndianCPU2 || 0)
datadoubleword← GPR[rt]63-8*bytesel..0 || 08*bytesel
StoreMemory (CCA, HALFWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

Shift Word Left Logical SLL



Format: `SLL rd, rt, sa` **MIPS32**

Purpose:

To left-shift a word by a fixed number of bits

Description: `GPR[rd] ← GPR[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 || 0^s
GPR[rd]← sign_extend(temp)
```

Exceptions:

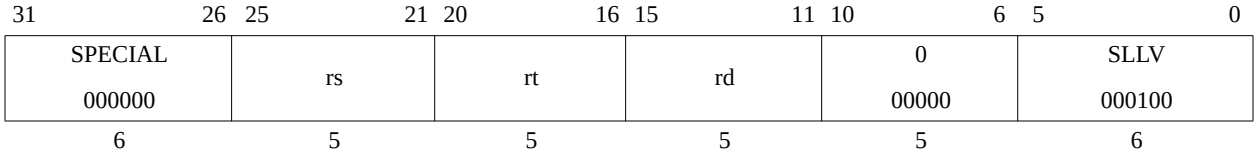
None

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.



Format: `SLLV rd, rt, rs` **MIPS32**

Purpose:

To left-shift a word by a variable number of bits

Description: `GPR[rd] ← GPR[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:

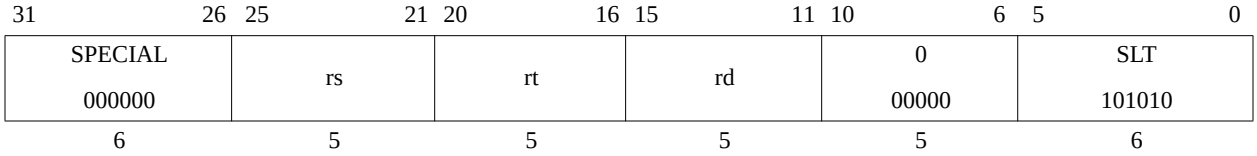
```
s       ← GPR[rs]4..0
temp    ← GPR[rt](31-s)..0 || 0s
GPR[rd]← sign_extend(temp)
```

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.

Set on Less Than SLT



Format: `SLT rd, rs, rt` **MIPS32**

Purpose:

To record the result of a less-than comparison

Description: `GPR[rd] ← (GPR[rs] < GPR[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

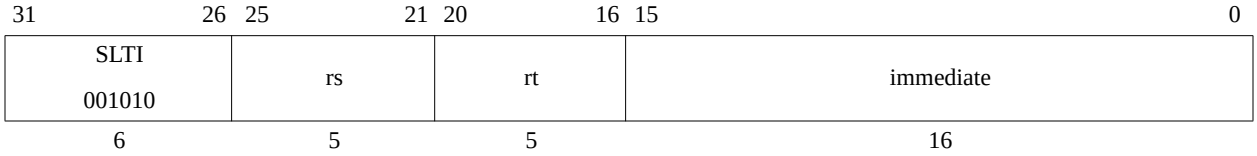
Operation:

```
if GPR[rs] < GPR[rt] then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

Set on Less Than Immediate SLTI



Format: `SLTI rt, rs, immediate` **MIPS32**

Purpose:

To record the result of a less-than comparison with a constant

Description: `GPR[rt] ← (GPR[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

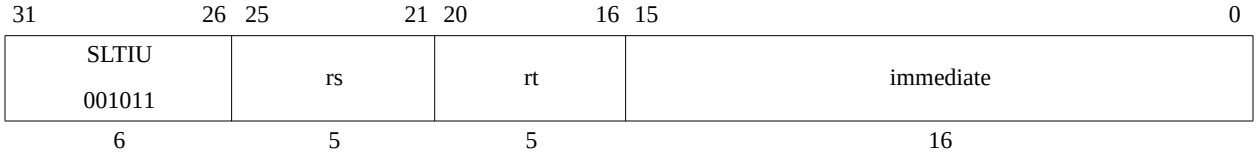
Operation:

```
if GPR[rs] < sign_extend(immediate) then
   GPR[rt] ← 0^(GPRLEN-1) || 1
else
   GPR[rt] ← 0^GPRLEN
endif
```

Exceptions:

None

Set on Less Than Immediate Unsigned — SLTIU



Format: `SLTIU rt, rs, immediate` **MIPS32**

Purpose:

To record the result of an unsigned less-than comparison with a constant

Description: `GPR[rt] ← (GPR[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

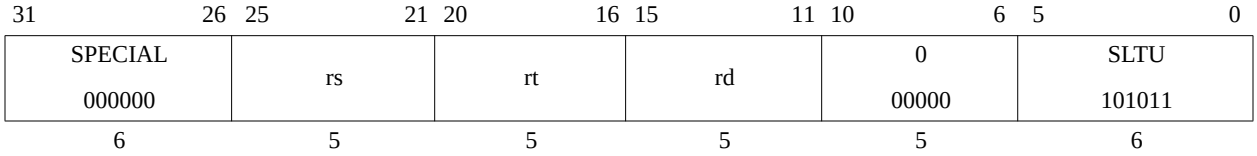
Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
   GPR[rt] ← 0^(GPRLEN-1) || 1
else
   GPR[rt] ← 0^GPRLEN
endif
```

Exceptions:

None

Set on Less Than Unsigned — SLTU



Format: `SLTU rd, rs, rt` **MIPS32**

Purpose:

To record the result of an unsigned less-than comparison

Description: `GPR[rd] ← (GPR[rs] < GPR[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

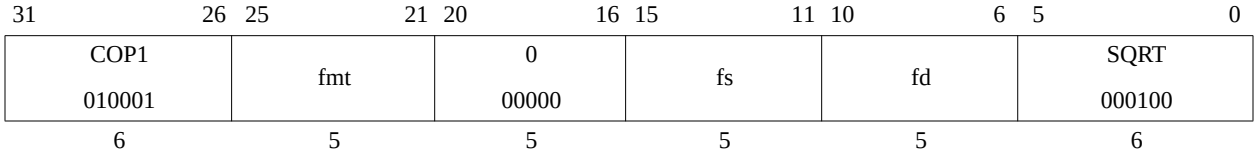
Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

Floating Point Square Root SQRT.fmt



Format: `SQRT.S fd, fs` **MIPS32**

`SQRT.D fd, fs` **MIPS32**

Purpose:

To compute the square root of an FP value

Description: `FPR[fd] ← SQRT(FPR[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 **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, fmt, SquareRoot(ValueFPR(fs, fmt)))
```

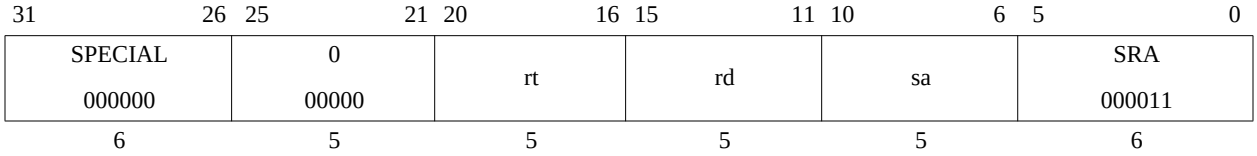
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Inexact, Unimplemented Operation

Shift Word Right Arithmetic SRA



Format: `SRA rd, rt, sa` **MIPS32**

Purpose:

To execute an arithmetic right-shift of a word by a fixed number of bits

Description: `GPR[rd] ← GPR[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:

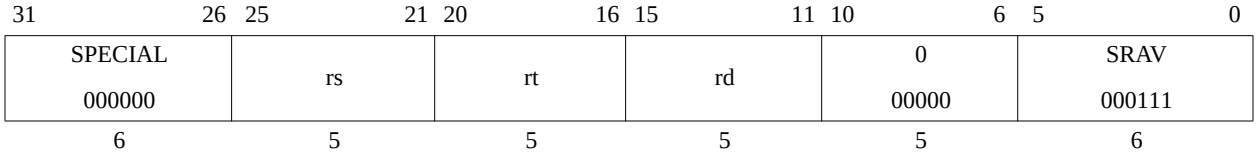
On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if NotWordValue(GPR[rt]) then
   UNPREDICTALBE
endif
s      ← sa
temp   ← (GPR[rt]31)^s || GPR[rt]31..s
GPR[rd]← sign_extend(temp)
```

Exceptions: None

Shift Word Right Arithmetic Variable SRAV



Format: `SRAV rd, rt, rs` **MIPS32**

Purpose:

To execute an arithmetic right-shift of a word by a variable number of bits

Description: `GPR[rd] ← GPR[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.

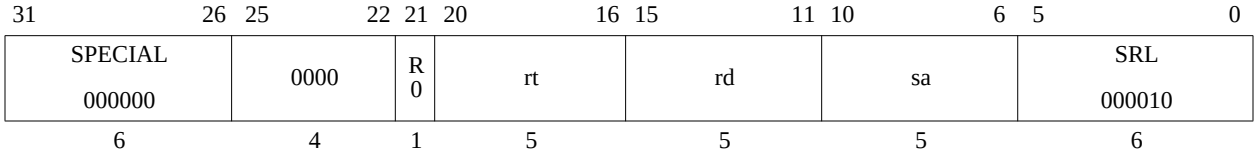
Operation:

```
if NotWordValue(GPR[rt]) then
   UNPREDICTABLE
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



Format: `SRL rd, rt, sa` **MIPS32**

Purpose:

To execute a logical right-shift of a word by a fixed number of bits

Description: `GPR[rd] ← GPR[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.

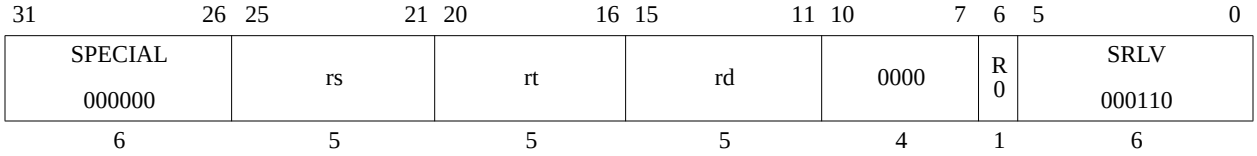
Operation:

```
if NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
s      ← sa
temp   ← 0^s || GPR[rt]_31..s
GPR[rd]← sign_extend(temp)
```

Exceptions:

None

Shift Word Right Logical Variable SRLV



Format: `SRLV rd, rt, rs` **MIPS32**

Purpose:

To execute a logical right-shift of a word by a variable number of bits

Description: `GPR[rd] ← GPR[rt] >> GPR[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:

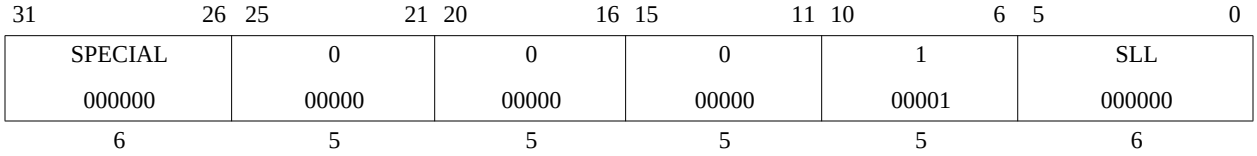
On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if NotWordValue(GPR[rt]) then
   UNPREDICTABLE
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:

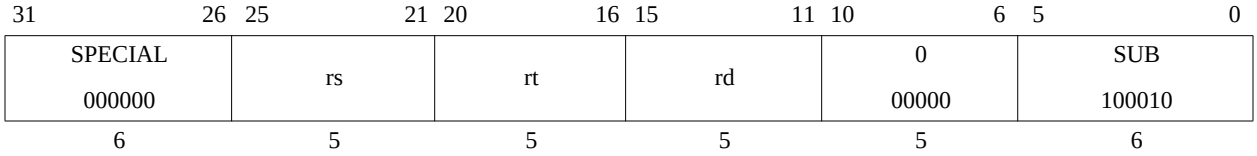
SSNOP is intended for use primarily to allow the programmer control over CP0 hazards by converting instructions into cycles in a superscalar processor. For example, to insert at least two cycles between an MTC0 and an ERET, one would use the following sequence:

```
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



Format: `SUB rd, rs, rt` **MIPS32**

Purpose:

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

Description: `GPR[rd] ← GPR[rs] - GPR[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:

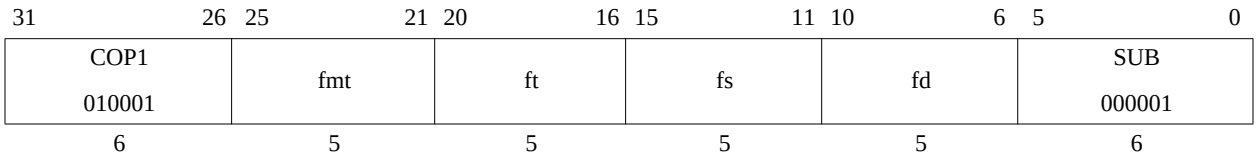
```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← (GPR[rs]31||GPR[rs]31..0) – (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← sign_extend(temp31..0)
endif
```

Exceptions:

Integer Overflow

Programming Notes:

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



Format: `SUB.S fd, fs, ft` **MIPS32**

`SUB.D fd, fs, ft` **MIPS32**

`SUB.PS fd, fs, ft` **MIPS64, MIPS32 Release 2**

Purpose:

To subtract FP values

Description: `FPR[fd] ← FPR[fs] - FPR[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))
```

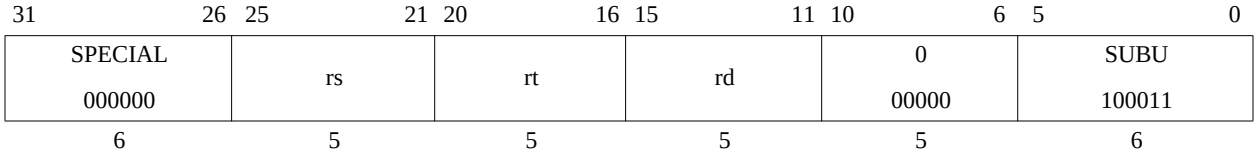
CPU Exceptions:

Coprocessor Unusable, Reserved Instruction

FPU Exceptions:

Inexact, Overflow, Underflow, Invalid Op, Unimplemented Op

Subtract Unsigned Word SUBU



Format: `SUBU rd, rs, rt` **MIPS32**

Purpose:

To subtract 32-bit integers

Description: `GPR[rd] ← GPR[rs] - GPR[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
   UNPREDICTABLE
endif
temp   ← GPR[rs] - GPR[rt]
GPR[rd]← sign_extend(temp)
```

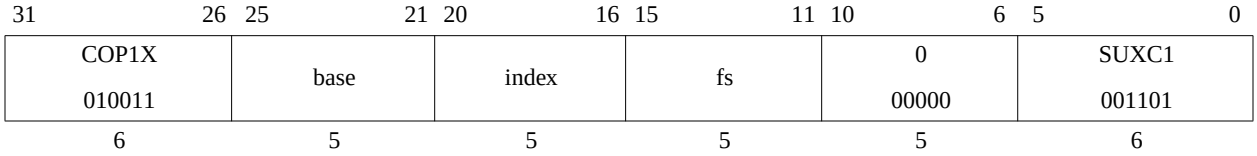
Exceptions:

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



Format: `SUXC1 fs, index(base)` **MIPS64, MIPS32 Release 2**

Purpose:

To store a doubleword from an FPR to memory (GPR+GPR addressing) ignoring alignment

Description: `memory[(GPR[base] + GPR[index])`$_{\texttt{PSIZE-1..3}}$`] ← FPR[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 doubleword-aligned; $EffectiveAddress_{2..0}$ are ignored.

Restrictions:

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

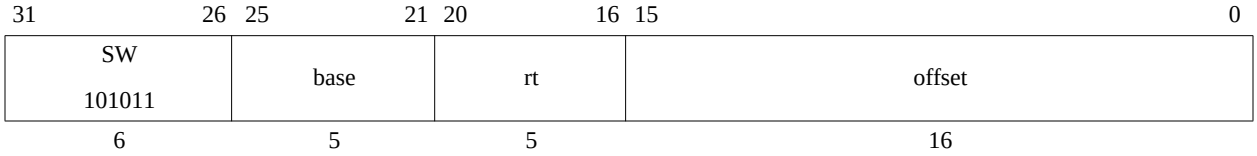
Operation:

```
vAddr ← (GPR[base]+GPR[index])63..3  || 03
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Watch

Store Word SW



Format: `SW rt, offset(base)` **MIPS32**

Purpose:

To store a word to memory

Description: `memory[GPR[base] + offset] ← GPR[rt]`

The least-significant 32-bit word of 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 either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

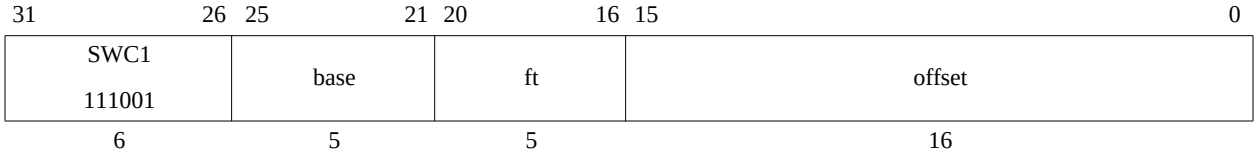
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
bytesel← vAddr2..0 xor (BigEndianCPU || 0^2)
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, Watch

Store Word from Floating Point — SWC1



Format: `SWC1 ft, offset(base)` **MIPS32**

Purpose:

To store a word from an FPR to memory

Description: `memory[GPR[base] + offset] ← FPR[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).

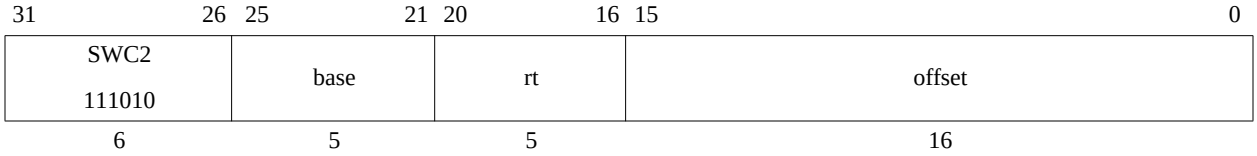
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^3 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
bytesel← vAddr2..0 xor (BigEndianCPU || 0^2)
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, Watch

Store Word from Coprocessor 2 — SWC2



Format: `SWC2 rt, offset(base)` **MIPS32**

Purpose:

To store a word from a COP2 register to memory

Description: `memory[GPR[base] + offset] ← CPR[2,rt,0]`

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).

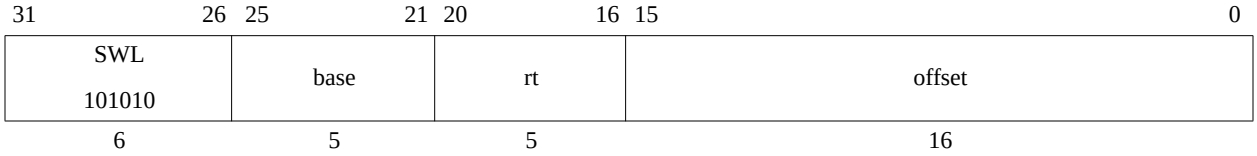
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 0^3 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
bytesel ← vAddr2..0 xor (BigEndianCPU || 0^2)
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, Watch

Store Word Left — SWL



Format: `SWL rt, offset(base)` **MIPS32**

Purpose:

To store the most-significant part of a word to an unaligned memory address

Description: `memory[GPR[base] + offset] ← GPR[rt]`

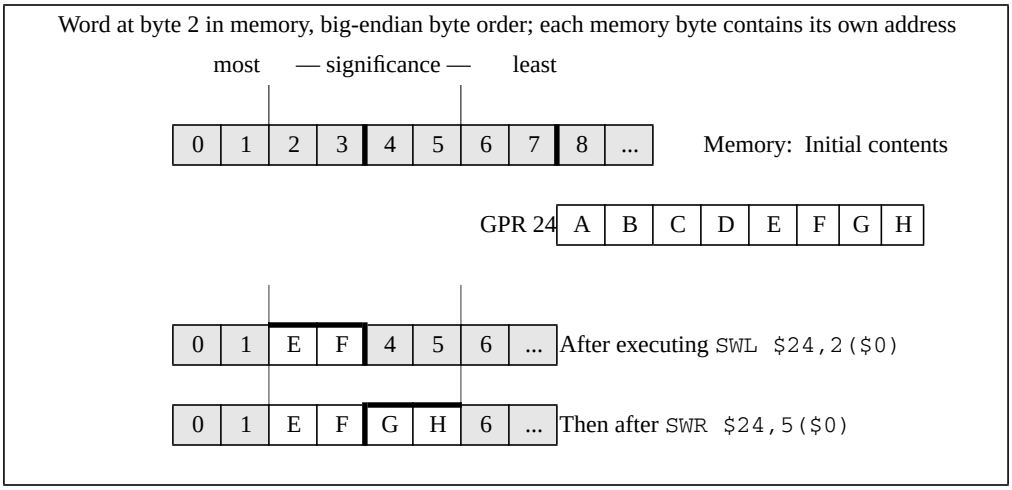
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

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.

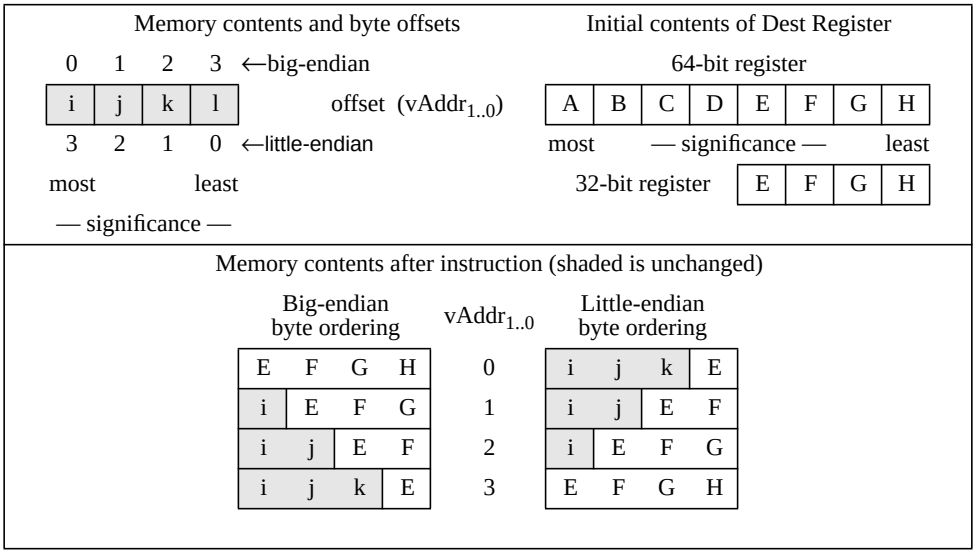
The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWL stores 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 3-23 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 3-24 Bytes Stored by an SWL Instruction



Restrictions:

None

Operation:

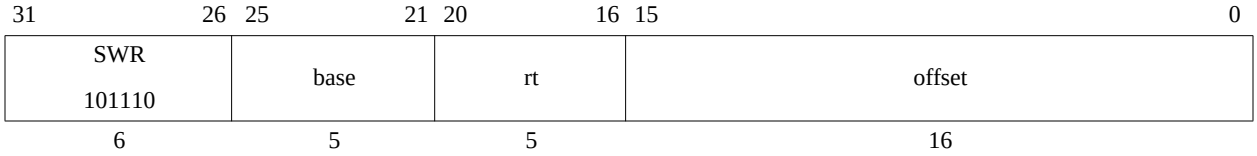
```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0  xor  ReverseEndian^3)
If BigEndianMem = 0 then
   pAddr ← pAddr_PSIZE-1..2 || 0^2
endif
byte   ← vAddr_1..0 xor BigEndianCPU^2
if (vAddr_2 xor BigEndianCPU) = 0 then
   datadoubleword ← 0^32 || 0^24-8*byte || GPR[rt]_31..24-8*byte
else
   datadoubleword ← 0^24-8*byte || GPR[rt]_31..24-8*byte || 0^32
endif

StoreMemory(CCA, byte, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error , Watch

Store Word Right SWR



Format: `SWR rt, offset(base)` **MIPS32**

Purpose:

To store the least-significant part of a word to an unaligned memory address

Description: `memory[GPR[base] + offset] ← GPR[rt]`

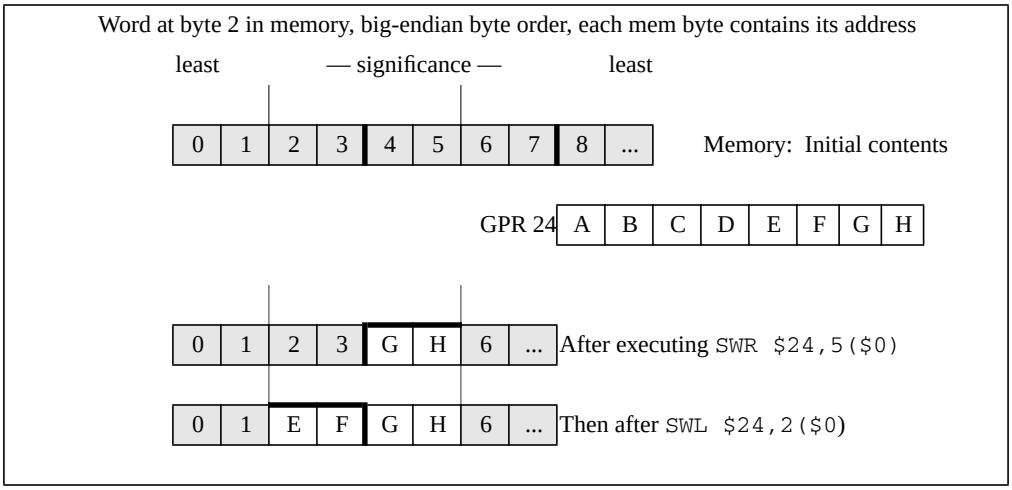
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

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.

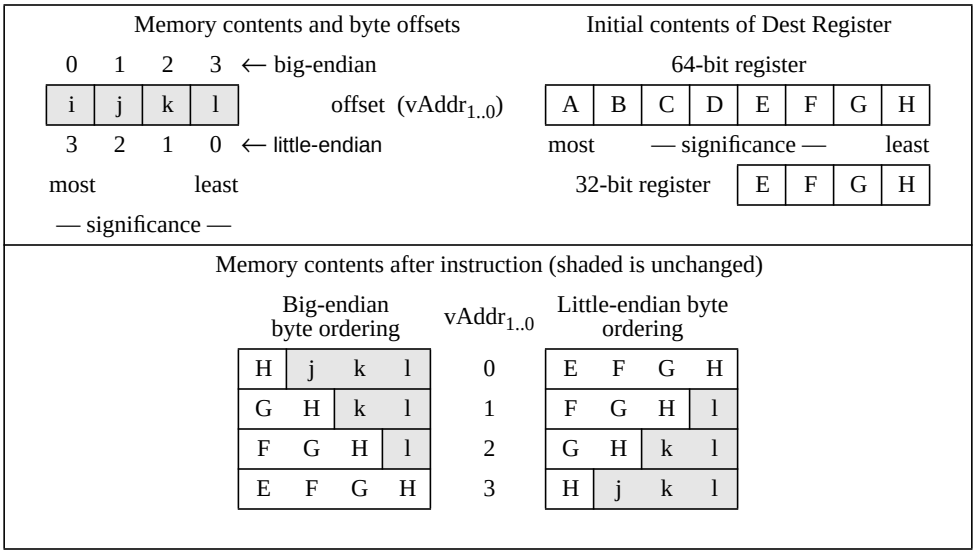
The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR 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 3-25 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 3-26 Bytes Stored by SWR Instruction



Restrictions:

None

Operation:

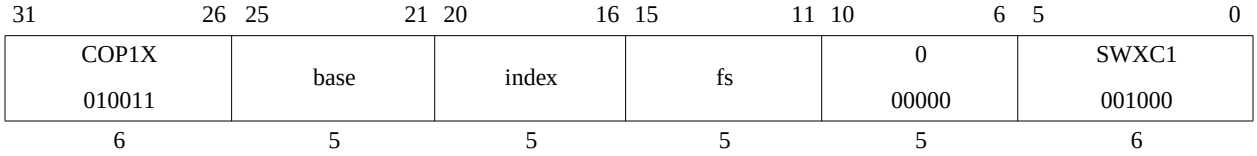
```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor ReverseEndian^3)
If BigEndianMem = 0 then
   pAddr ← pAddr_PSIZE-1..2 || 0^2
endif
byte   ← vAddr_1..0 xor BigEndianCPU^2
if (vAddr_2 xor BigEndianCPU) = 0 then
   datadoubleword ← 0^32 || GPR[rt]_31-8*byte..0 || 0^8*byte
else
   datadoubleword ← GPR[rt]_31-8*byte..0 || 0^8*byte || 0^32
endif

StoreMemory(CCA, WORD-byte, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch

Store Word Indexed from Floating Point — SWXC1



Format: `SWXC1 fs, index(base)` **MIPS64**
MIPS32 Release 2

Purpose:

To store a word from an FPR to memory (GPR+GPR addressing)

Description: `memory[GPR[base] + GPR[index]] ← FPR[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:

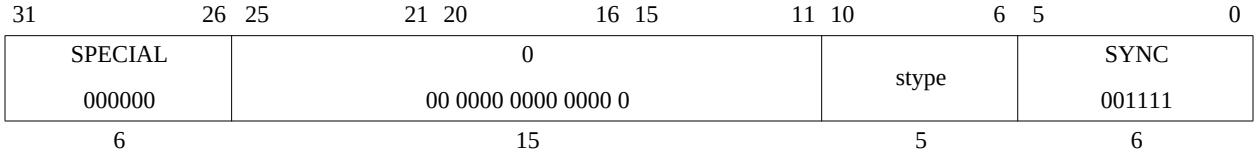
An Address Error exception occurs if EffectiveAddress$_{1..0} \neq 0$ (not word-aligned).

Operation:

```
vAddr ← GPR[base] + GPR[index]
if vAddr1..0 ≠ 0^3 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
bytesel← vAddr2..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, Watch



Format: `SYNC (stype = 0 implied)` **MIPS32**

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 (in Release 1 of the Architecture) or EHB (in Release 2 of the Architecture), 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 for future extensions to the architecture. A value of zero will always be defined such that it performs all defined synchronization operations. Non-zero values may be defined to remove some synchronization operations. As such, software should never use a non-zero value of the *stype* field, as this may inadvertently cause future failures if non-zero values remove synchronization operations.

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 **UNPREDICTABLE** 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.

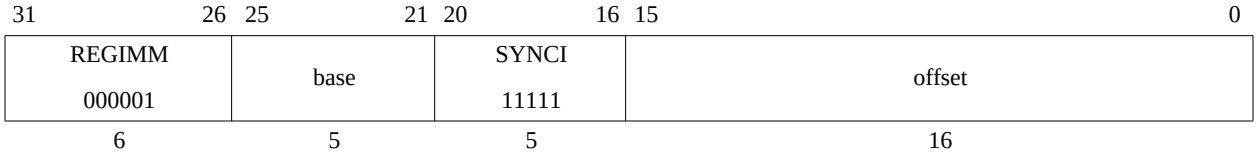
```
# Processor A (writer)
# Conditions at entry:
# The value 0 has been stored in FLAG and that value is observable by B
SW     R1, DATA         # change shared DATA value
LI     R2, 1
SYNC                    # Perform DATA store before performing FLAG store
SW     R2, FLAG         # say that the shared 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.

Synchronize Caches to Make Instruction Writes Effective — SYNCI



Format: `SYNCI offset(base)` **MIPS32 Release 2**

Purpose:

To synchronize all caches to make instruction writes effective.

Description:

This instruction is used after a new instruction stream is written to make the new instructions effective relative to an instruction fetch, when used in conjunction with the SYNC and JALR.HB, JR.HB, or ERET instructions, as described below. Unlike the CACHE instruction, the SYNCI instruction is available in all operating modes in an implementation of Release 2 of the architecture.

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 to address the cache line in all caches which may need to be synchronized with the write of the new instructions. The operation occurs only on the cache line which may contain the effective address. One SYNCI instruction is required for every cache line that was written. See the Programming Notes below.

A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur as a byproduct of this instruction. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS.

A Cache Error exception may occur as a byproduct of 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 SYNCI instruction whose address matches the Watch register address match conditions.

Restrictions:

The operation of the processor is **UNPREDICTABLE** if the effective address references any instruction cache line that contains instructions to be executed between the SYNCI and the subsequent JALR.HB, JR.HB, or ERET instruction required to clear the instruction hazard.

The SYNCI instruction has no effect on cache lines that were prevsiously locked with the CACHE instruction. If correct software operation depends on the state of a locked line, the CACHE instruction must be used to synchronize the caches.

The SYNCI instruction acts only on the current processor. It doesn't not affect the caches on other processors in a multi-processor system, except as required to perform the operation on the current processor (as might be the case if multiple processors share an L2 or L3 cache).

Full visibility of the new instruction stream requires execution of a subsequent SYNC instruction, followed by a JALR.HB, JR.HB, DERET, or ERET instruction. The operation of the processor is **UNPREDICTABLE** if this sequence is not followed.

Operation:

```
vaddr ← GPR[base] + sign_extend(offset)
SynchronizeCacheLines(vaddr)/* Operate on all caches */
```

Exceptions:

Reserved Instruction Exception (Release 1 implementations only)
TLB Refill Exception
TLB Invalid Exception
Address Error Exception
Cache Error Exception
Bus Error Exception

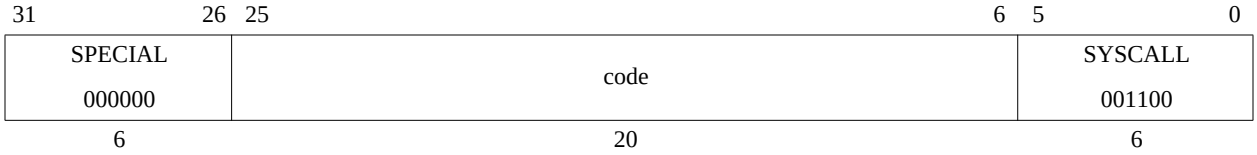
Programming Notes:

When the instruction stream is written, the SYNCI instruction should be used in conjunction with other instructions to make the newly-written instructions effective. The following example shows a routine which can be called after the new instruction stream is written to make those changes effective. Note that the SYNCI instruction could be replaced with the corresponding sequence of CACHE instructions (when access to Coprocessor 0 is available), and that the JR.HB instruction could be replaced with JALR.HB, ERET, or DERET instructions, as appropriate. A SYNC instruction is required between the final SYNCI instruction in the loop and the instruction that clears instruction hazards.

```
/*
 * This routine makes changes to the instruction stream effective to the
 * hardware.  It should be called after the instruction stream is written.
 * On return, the new instructions are effective.
 *
 * Inputs:
 *     a0 = Start address of new instruction stream
 *     a1 = Size, in bytes, of new instruction stream
 */

   addu   a1, a0, a1          /* Calculate end address + 1 */
                              /* (daddu for 64-bit addressing) */
   rdhwr  v0, HW_SYNCI_Step   /* Get step size for SYNCI from new */
                              /*   Release 2 instruction */
   beq    v0, zero, 20f       /* If no caches require synchronization, */
   nop                        /*   branch around */
10:synci  0(a0)               /* Synchronize all caches around address */
   sltu   v1, a0, a1          /* Compare current with end address */
   bne    v1, zero, 10b       /* Branch if more to do */
   addu   a0, a0, v0          /* Add step size in delay slot */
                              /* (daddu for 64-bit addressing) */
   sync                       /* Clear memory hazards */
20:jr.hb  ra                  /* Return, clearing instruction hazards */
   nop
```

System Call SYSCALL



Format: `SYSCALL` **MIPS32**

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

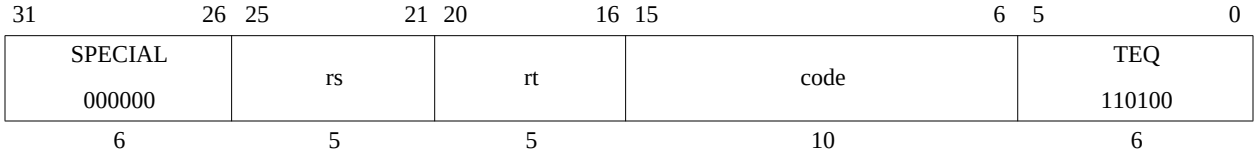
Operation:

```
SignalException(SystemCall)
```

Exceptions:

System Call

Trap if Equal — TEQ



Format: `TEQ rs, rt` **MIPS32**

Purpose:

To compare GPRs and do a conditional trap

Description: `if GPR[rs] = GPR[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

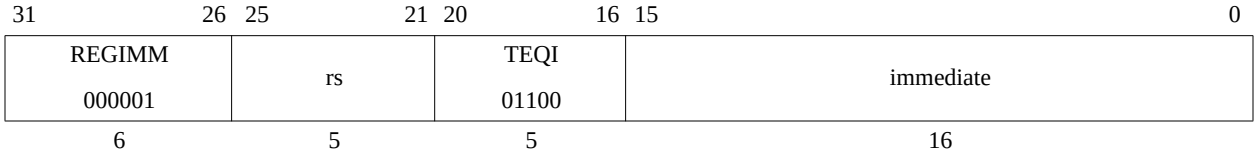
Operation:

```
if GPR[rs] = GPR[rt] then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Trap if Equal Immediate TEQI



Format: `TEQI rs, immediate` **MIPS32**

Purpose:

To compare a GPR to a constant and do a conditional trap

Description: `if GPR[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

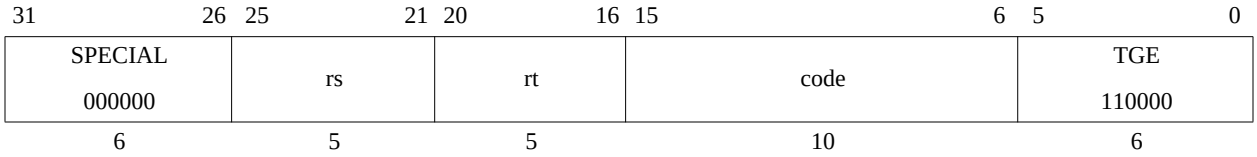
Operation:

```
if GPR[rs] = sign_extend(immediate) then
       SignalException(Trap)
endif
```

Exceptions:

Trap

Trap if Greater or Equal — TGE



Format: `TGE rs, rt` **MIPS32**

Purpose:

To compare GPRs and do a conditional trap

Description: `if GPR[rs] ≥ GPR[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

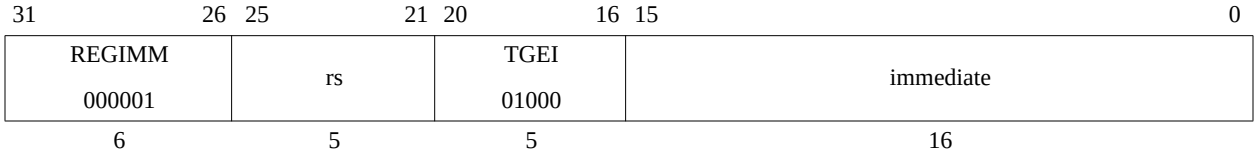
Operation:

```
if GPR[rs] ≥ GPR[rt] then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Trap if Greater or Equal Immediate TGEI



Format: `TGEI rs, immediate` **MIPS32**

Purpose:

To compare a GPR to a constant and do a conditional trap

Description: `if GPR[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

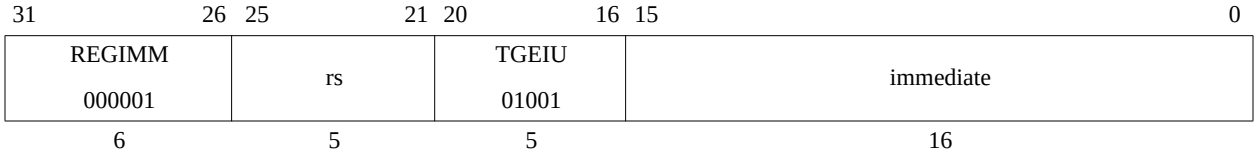
Operation:

```
if GPR[rs] ≥ sign_extend(immediate) then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Trap if Greater or Equal Immediate Unsigned — TGEIU



Format: `TGEIU rs, immediate` **MIPS32**

Purpose:

To compare a GPR to a constant and do a conditional trap

Description: `if GPR[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

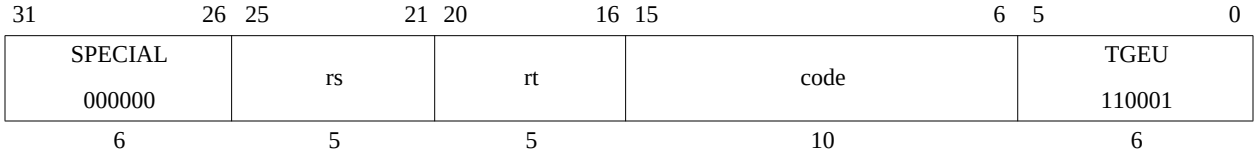
Operation:

```
if (0 || GPR[rs]) ≥ (0 || sign_extend(immediate)) then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Trap if Greater or Equal Unsigned — TGEU



Format: `TGEU rs, rt` **MIPS32**

Purpose:

To compare GPRs and do a conditional trap

Description: `if GPR[rs] ≥ GPR[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:

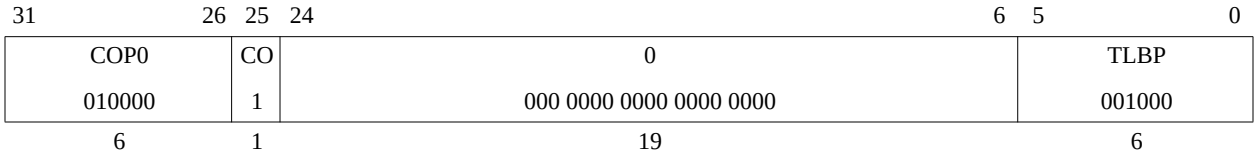
None

Operation:

```
if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
   SignalException(Trap)
endif
```

Exceptions:

Trap



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. In Release 1 of the Architecture, it is implementation dependent whether multiple TLB matches are detected on a TLBP. However, implementations are strongly encouraged to report multiple TLB matches only on a TLB write. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write.

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

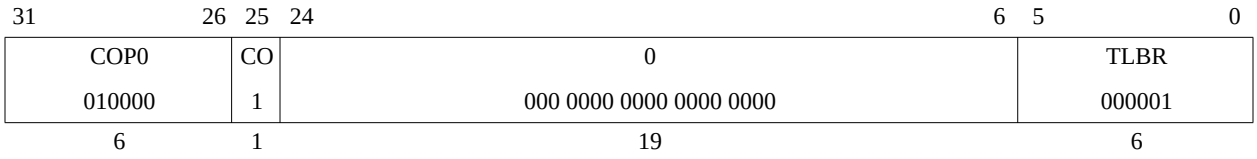
Operation:

```
Index ← 1 || UNPREDICTABLE^31
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

Machine Check



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. In Release 1 of the Architecture, it is implementation dependent whether multiple TLB matches are detected on a TLBR. However, implementations are strongly encouraged to report multiple TLB matches only on a TLB write. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. 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.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

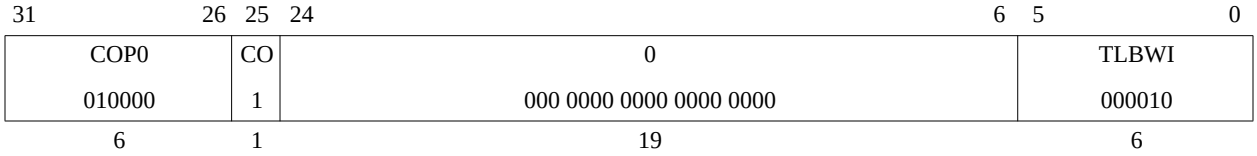
```
i ← Index
if i > (TLBEntries - 1) then
   UNDEFINED
endif
PageMask_Mask ← TLB[i]_Mask
EntryHi ← TLB[i]_R || 0^Fill ||
          (TLB[i]_VPN2 and not TLB[i]_Mask) || # Masking implementation dependent
          0^5 || TLB[i]_ASID
EntryLo1 ← 0^Fill ||
          (TLB[i]_PFN1 and not TLB[i]_Mask) || # 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

Machine Check

Write Indexed TLB Entry — TLBWI



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. It is implementation dependent whether multiple TLB matches are detected on a TLBWI. In such an instance, a Machine Check Exception is signaled. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. 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.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

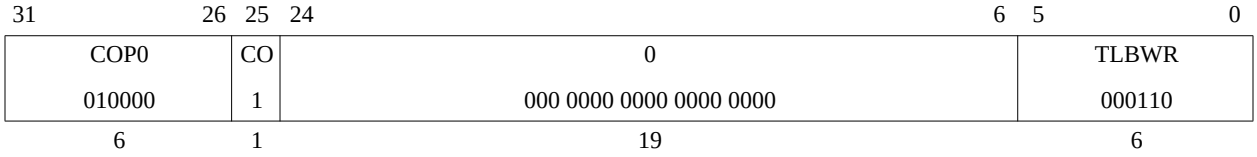
```
i ← Index
TLB[i]Mask ← PageMaskMask
TLB[i]R ← EntryHiR
TLB[i]VPN2 ← EntryHiVPN2 and not PageMaskMask # Implementation dependent
TLB[i]ASID ← EntryHiASID
TLB[i]G ← EntryLo1G and EntryLo0G
TLB[i]PFN1 ← EntryLo1PFN and not PageMaskMask # Implementation dependent
TLB[i]C1 ← EntryLo1C
TLB[i]D1 ← EntryLo1D
TLB[i]V1 ← EntryLo1V
TLB[i]PFN0 ← EntryLo0PFN and not PageMaskMask # Implementation dependent
TLB[i]C0 ← EntryLo0C
TLB[i]D0 ← EntryLo0D
TLB[i]V0 ← EntryLo0V
```

Exceptions:

Coprocessor Unusable

Machine Check

Write Random TLB Entry — TLBWR



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. It is implementation dependent whether multiple TLB matches are detected on a TLBWR. In such an instance, a Machine Check Exception is signaled. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. 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:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

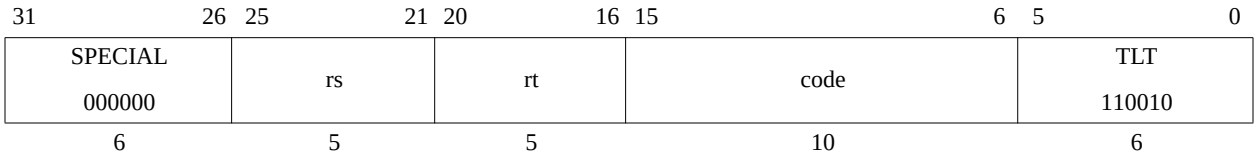
```
i ← Random
TLB[i]Mask ← PageMaskMask
TLB[i]R ← EntryHiR
TLB[i]VPN2 ← EntryHiVPN2 and not PageMaskMask # Implementation dependent
TLB[i]ASID ← EntryHiASID
TLB[i]G ← EntryLo1G and EntryLo0G
TLB[i]PFN1 ← EntryLo1PFN and not PageMaskMask # Implementation dependent
TLB[i]C1 ← EntryLo1C
TLB[i]D1 ← EntryLo1D
TLB[i]V1 ← EntryLo1V
TLB[i]PFN0 ← EntryLo0PFN and not PageMaskMask # Implementation dependent
TLB[i]C0 ← EntryLo0C
TLB[i]D0 ← EntryLo0D
TLB[i]V0 ← EntryLo0V
```

Exceptions:

Coprocessor Unusable

Machine Check

Trap if Less Than TLT



Format: `TLT rs, rt` **MIPS32**

Purpose:

To compare GPRs and do a conditional trap

Description: `if GPR[rs] < GPR[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

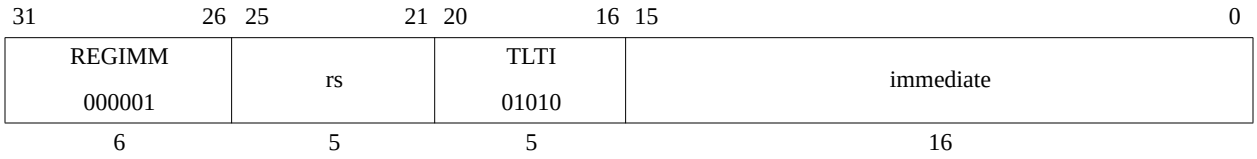
Operation:

```
if GPR[rs] < GPR[rt] then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Trap if Less Than Immediate TLTI



Format: `TLTI rs, immediate` **MIPS32**

Purpose:

To compare a GPR to a constant and do a conditional trap

Description: `if GPR[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

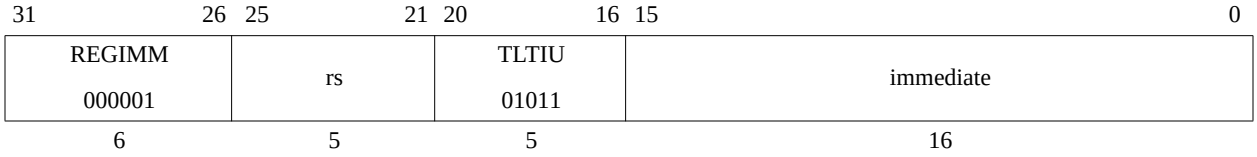
Operation:

```
if GPR[rs] < sign_extend(immediate) then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Trap if Less Than Immediate Unsigned — TLTIU



Format: `TLTIU rs, immediate` **MIPS32**

Purpose:

To compare a GPR to a constant and do a conditional trap

Description: `if GPR[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

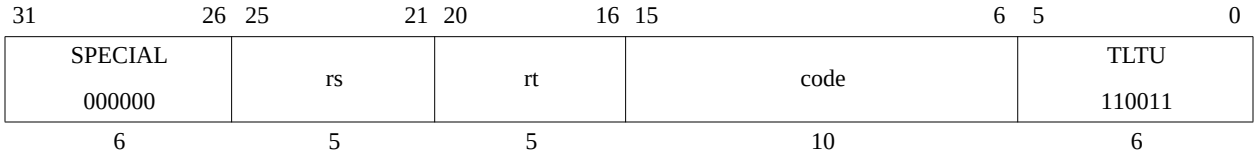
Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Trap if Less Than Unsigned TLTU



Format: `TLTU rs, rt` **MIPS32**

Purpose:

To compare GPRs and do a conditional trap

Description: `if GPR[rs] < GPR[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:

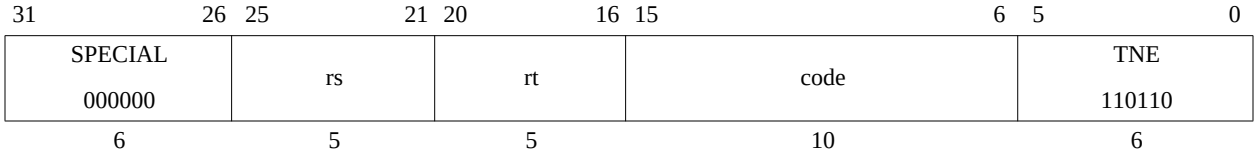
```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Trap if Not Equal

TNE



Format: `TNE rs, rt` **MIPS32**

Purpose:

To compare GPRs and do a conditional trap

Description: `if GPR[rs] ≠ GPR[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

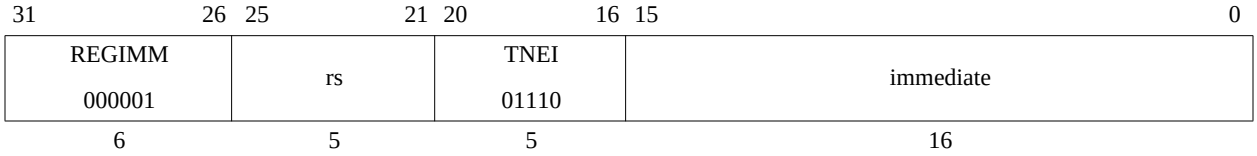
Operation:

```
if GPR[rs] ≠ GPR[rt] then
       SignalException(Trap)
endif
```

Exceptions:

Trap

Trap if Not Equal Immediate TNEI



Format: `TNEI rs, immediate` **MIPS32**

Purpose:

To compare a GPR to a constant and do a conditional trap

Description: `if GPR[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:

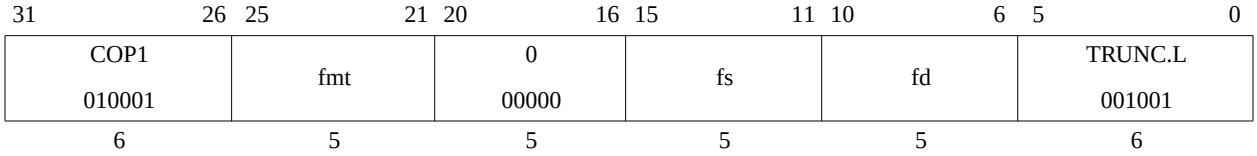
None

Operation:

```
if GPR[rs] ≠ sign_extend(immediate) then
   SignalException(Trap)
endif
```

Exceptions:

Trap



Format: `TRUNC.L.S fd, fs` **MIPS64, MIPS32 Release 2**

`TRUNC.L.D fd, fs` **MIPS64, MIPS32 Release 2**

Purpose:

To convert an FP value to 64-bit fixed point, rounding toward zero

Description: `FPR[fd] ← convert_and_round(FPR[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))
```

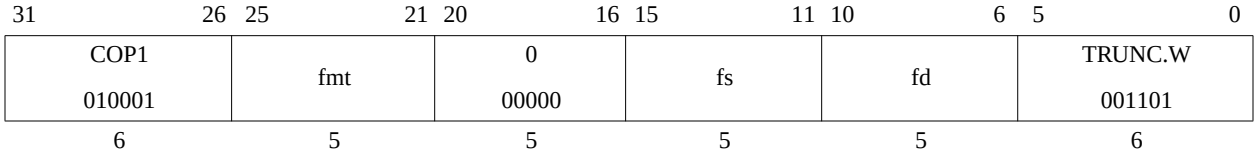
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Overflow, Inexact

Floating Point Truncate to Word Fixed Point — TRUNC.W.fmt



Format: `TRUNC.W.S fd, fs` **MIPS32**

`TRUNC.W.D fd, fs` **MIPS32**

Purpose:

To convert an FP value to 32-bit fixed point, rounding toward zero

Description: `FPR[fd] ← convert_and_round(FPR[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))
```

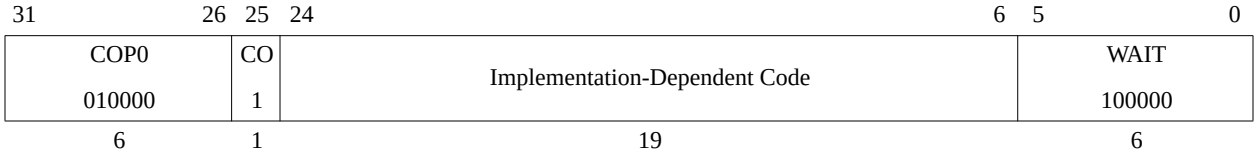
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Invalid Operation, Overflow, Unimplemented Operation

Enter Standby Mode — WAIT



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.The assertion of any reset or NMI must restart the pipeline and the corresponding exception must be taken.

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).

Restrictions:

The operation of the processor is **UNDEFINED** if a WAIT instruction is placed in the delay slot of a branch or a jump.

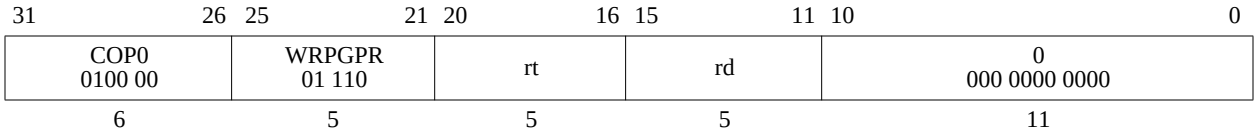
If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
I: Enter implementation dependent lower power mode
I+1:/* Potential interrupt taken here */
```

Exceptions:

Coprocessor Unusable Exception



Format: `WRPGPR rd, rt` **MIPS32 Release 2**

Purpose:

To move the contents of a current GPR to a GPR in the previous shadow set.

Description: `SGPR[SRSCtl`$_{PSS}$`, rd] ← GPR[rt]`

The contents of the current GPR *rt* is moved to the shadow GPR register specified by $SRSCtl_{PSS}$ (signifying the previous shadow set number) and *rd* (specifying the register number within that set).

Restrictions:

In implementations prior to Release 2 of the Architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

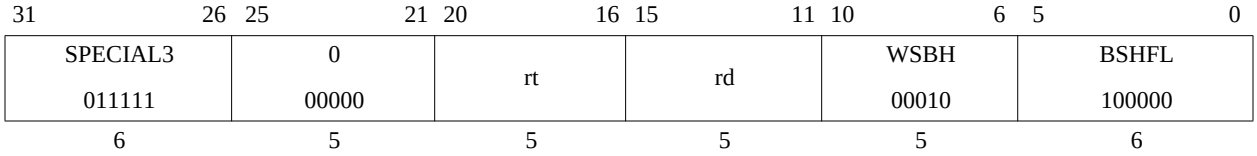
```
SGPR[SRSCtl_PSS, rd] ← GPR[rt]
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Word Swap Bytes Within Halfwords — WSBH



Format: `wsbh rd, rt` **MIPS32 Release 2**

Purpose:

To swap the bytes within each halfword of GPR *rt* and store the value into GPR *rd*.

Description: `GPR[rd] ← SwapBytesWithinHalfwords(GPR[rt])`

Within each halfword of the lower word of GPR *rt* the bytes are swapped, the result is sign-extended, and stored in GPR *rd*.

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

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[rt]) then
   UNPREDICTABLE
endif
GPR[rd] ←sign_extend(GPR[rt]23..16 || GPR[rt]31..24 || GPR[rt]7..0 || GPR[rt]15..8)
```

Exceptions:

Reserved Instruction

Programming Notes:

The WSBH instruction can be used to convert halfword and word data of one endianness to another endianness. The endianness of a word value can be converted using the following sequence:

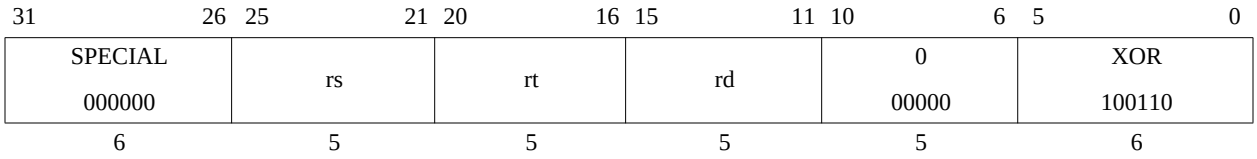
```
lw     t0, 0(a1)           /* Read word value */
wsbh   t0, t0              /* Convert endiannes of the halfwords */
rotr   t0, t0, 16          /* Swap the halfwords within the words */
```

Combined with SEH and SRA, two contiguous halfwords can be loaded from memory, have their endianness converted, and be sign-extended into two word values in four instructions. For example:

```
lw     t0, 0(a1)           /* Read two contiguous halfwords */
wsbh   t0, t0              /* Convert endiannes of the halfwords */
seh    t1, t0              /* t1 = lower halfword sign-extended to word */
sra    t0, t0, 16          /* t0 = upper halfword sign-extended to word */
```

Zero-extended words can be created by changing the SEH and SRA instructions to ANDI and SRL instructions, respectively.

Exclusive OR XOR



Format: `XOR rd, rs, rt` **MIPS32**

Purpose:

To do a bitwise logical Exclusive OR

Description: `GPR[rd] ← GPR[rs] XOR GPR[rt]`

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical Exclusive OR operation and place the result into GPR *rd*.

Restrictions:

None

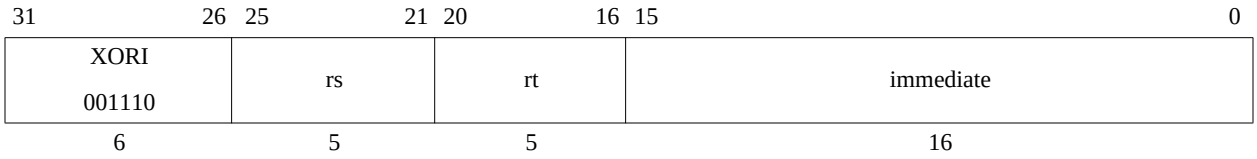
Operation:

```
GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None

Exclusive OR Immediate XORI



Format: `XORI rt, rs, immediate` **MIPS32**

Purpose:

To do a bitwise logical Exclusive OR with a constant

Description: `GPR[rt] ← GPR[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

Instruction Bit Encodings

A.1 Instruction Encodings and Instruction Classes

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

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

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

A.2 Instruction Bit Encoding Tables

This section provides various bit encoding tables for the instructions of the MIPS64® ISA.

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

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

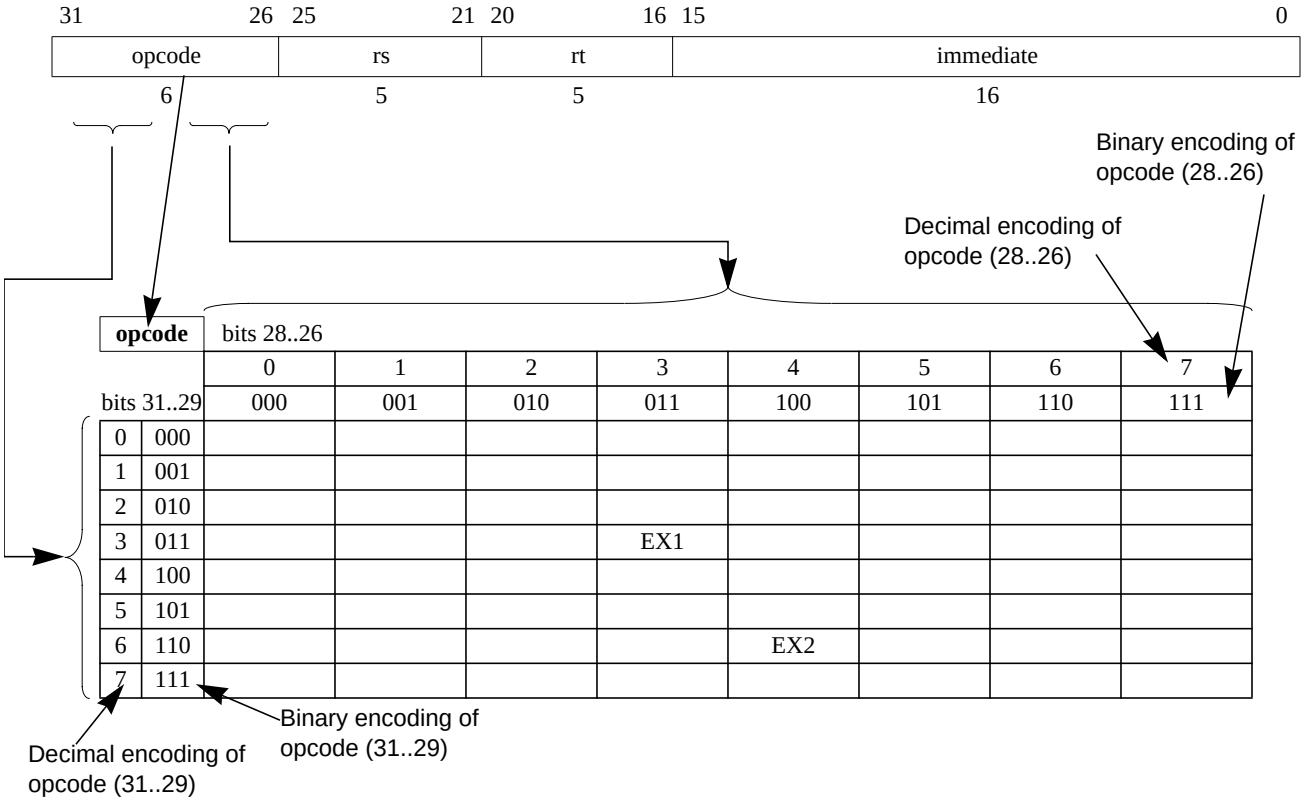


Figure A-1 Sample Bit Encoding Table

Tables A-2 through A-23 describe the encoding used for the MIPS64 ISA. Table A-1 describes the meaning of the symbols used in the tables.

Table A-1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction must cause a Reserved Instruction Exception.
δ	(Also *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
β	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level or a new revision of the Architecture. Executing such an instruction must cause a Reserved Instruction Exception.
⊥	Operation or field codes marked with this symbol represent instructions which are not legal if the processor is configured to be backward compatible with MIPS32 processors. If the processor is executing with 64-bit operations enabled, execution proceeds normally. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).

Table A-1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
∇	Operation or field codes marked with this symbol represent instructions which were only legal if 64-bit operations were enabled on implementations of Release 1 of the Architecture. In Release 2 of the architecture, operation or field codes marked with this symbol represent instructions which are legal if 64-bit floating point operations are enabled. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
θ	Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, MIPS Technologies will assist the partner in selecting appropriate encodings if requested by the partner. The partner is not required to consult with MIPS Technologies when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (*SPECIAL2* encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
σ	Field codes marked with this symbol represent an EJTAG support instruction and implementation of this encoding is optional for each implementation. If the encoding is not implemented, executing such an instruction must cause a Reserved Instruction Exception. If the encoding is implemented, it must match the instruction encoding as shown in the table.
ε	Operation or field codes marked with this symbol are reserved for MIPS Application Specific Extensions. If the ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception.
ϕ	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS64 ISA. Software should avoid using these operation or field codes.
⊕	Operation or field codes marked with this symbol are valid for Release 2 implementations of the architecture. Executing such an instruction in a Release 1 implementation must cause a Reserved Instruction Exception.

Table A-2 MIPS64 Encoding of the Opcode Field

opcode		*bits 28..26*							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0* δ	*COP1* δ	*COP2* θδ	*COP1X* δ	BEQL ϕ	BNEL ϕ	BLEZL ϕ	BGTZL ϕ
3	011	DADDI ⊥	DADDIU ⊥	LDL ⊥	LDR ⊥	*SPECIAL2* δ	JALX ε	*MDMX* εδ	*SPECIAL3*[1] δ⊕
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU ⊥
5	101	SB	SH	SWL	SW	SDL ⊥	SDR ⊥	SWR	CACHE
6	110	LL	LWC1	LWC2 θ	PREF	LLD ⊥	LDC1	LDC2 θ	LD ⊥
7	111	SC	SWC1	SWC2 θ	*	SCD ⊥	SDC1	SDC2 θ	SD ⊥

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

Table A-3 MIPS64 *SPECIAL* Opcode Encoding of Function Field

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL[1]	*MOVCI* δ	*SRL* δ	SRA	SLLV	*	*SRLV* δ	SRAV
1	001	JR[2]	JALR[2]	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	DSLLV ⊥	*	*DSRLV* δ⊥	DSRAV ⊥
3	011	MULT	MULTU	DIV	DIVU	DMULT ⊥	DMULTU ⊥	DDIV ⊥	DDIVU ⊥
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	DADD ⊥	DADDU ⊥	DSUB ⊥	DSUBU ⊥
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	DSLL ⊥	*	*DSRL* δ⊥	DSRA ⊥	DSLL32 ⊥	*	*DSRL32* δ⊥	DSRA32 ⊥

1. Specific encodings of the *rt*, *rd*, and *sa* fields are used to distinguish among the SLL, NOP, SSNOP and EHB functions.
2. Specific encodings of the hint field are used to distinguish JR from JR.HB and JALR from JALR.HB

Table A-4 MIPS64 *REGIMM* Encoding of rt Field

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

Table A-5 MIPS64 *SPECIAL2* Encoding of Function Field

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	θ	MSUB	MSUBU	θ	θ
1	001	θ	θ	θ	θ	θ	θ	θ	θ
2	010	θ	θ	θ	θ	θ	θ	θ	θ
3	011	θ	θ	θ	θ	θ	θ	θ	θ
4	100	CLZ	CLO	θ	θ	DCLZ ⊥	DCLO ⊥	θ	θ
5	101	θ	θ	θ	θ	θ	θ	θ	θ
6	110	θ	θ	θ	θ	θ	θ	θ	θ
7	111	θ	θ	θ	θ	θ	θ	θ	SDBBP σ

Table A-6 MIPS64 *SPECIAL3*[1] Encoding of Function Field for Release 2 of the Architecture

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	EXT ⊕	DEXTM ⊥⊕	DEXTU ⊥⊕	DEXT ⊥⊕	INS ⊕	DINSM ⊥⊕	DINSU ⊥⊕	DINS ⊥⊕
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	*BSHFL* ⊕δ	*	*	*	*DBSHFL* ⊥⊕δ	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	RDHWR ⊕	*	*	*	*

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

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

tf	*bit 16*	
	0	1
	MOVF	MOVT

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

R	*bit 21*	
	0	1
	SRL	ROTR

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

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

R	*bit 6*	
	0	1
	SRLV	ROTRV

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

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

R	*bit 6*	
	0	1
	DSRLV	DROTRV

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

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

R	*bit 21*	
	0	1
	DSRL	DROTR

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

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

R	*bit 21*	
	0	1
	DSRL32	DROTR32

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

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

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

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

Table A-14 MIPS64 *COP0* Encoding of rs Field

rs		*bits 23..21*							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC0	DMFC0 ⊥	*	*	MTC0	DMTC0 ⊥	*	*
1	01	*	*	RDPGPR ⊕	*MFMC0*[1] δ⊕	*	*	WRPGPR ⊕	*
2	10	C0 δ							
3	11								

1. Release 2 of the Architecture added the MFMC0 function, which is further decoded as the DI and EI instructions.

Table A-15 MIPS64 *COP0* Encoding of Function Field When rs=*CO*

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	TLBR	TLBWI	*	*	*	TLBWR	*
1	001	TLBP	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	ERET	*	*	*	*	*	*	DERET σ
4	100	WAIT	*	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Table A-16 MIPS64 *COP1* Encoding of rs Field

rs		*bits 23..21*							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC1	DMFC1 ⊥	CFC1	MFHC1 ⊕	MTC1	DMTC1 ⊥	CTC1	MTHC1 ⊕
1	01	*BC1* δ	*BC1ANY2* δε∇	*BC1ANY4* δε∇	*	*	*	*	*
2	10	*S* δ	*D* δ	*	*	*W* δ	*L* δ	*PS* δ	*
3	11	*	*	*	*	*	*	*	*

Table A-17 MIPS64 *COP1* Encoding of Function Field When rs=*S*

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L ∇	TRUNC.L ∇	CEIL.L ∇	FLOOR.L ∇	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	MOVCF δ	MOVZ	MOVN	*	RECIP ∇	RSQRT ∇	*
3	011	*	*	*	*	RECIP2 ε∇	RECIP1 ε∇	RSQRT1 ε∇	RSQRT2 ε∇
4	100	*	CVT.D	*	*	CVT.W	CVT.L ∇	CVT.PS∇	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F CABS.F ε∇	C.UN CABS.UN ε∇	C.EQ CABS.EQ ε∇	C.UEQ CABS.UEQ ε∇	C.OLT CABS.OLT ε∇	C.ULT CABS.ULT ε∇	C.OLE CABS.OLE ε∇	C.ULE CABS.ULE ε∇
7	111	C.SF CABS.SF ε∇	C.NGLE CABS.NGLE ε∇	C.SEQ CABS.SEQ ε∇	C.NGL CABS.NGL ε∇	C.LT CABS.LT ε∇	C.NGE CABS.NGE ε∇	C.LE CABS.LE ε∇	C.NGT CABS.NGT ε∇

Table A-18 MIPS64 *COP1* Encoding of Function Field When rs=*D*

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L ∇	TRUNC.L ∇	CEIL.L ∇	FLOOR.L ∇	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	MOVCF δ	MOVZ	MOVN	*	RECIP ∇	RSQRT ∇	*
3	011	*	*	*	*	RECIP2 ε∇	RECIP1 ε∇	RSQRT1 ε∇	RSQRT2 ε∇
4	100	CVT.S	*	*	*	CVT.W	CVT.L ∇	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F CABS.F ε∇	C.UN CABS.UN ε∇	C.EQ CABS.EQ ε∇	C.UEQ CABS.UEQ ε∇	C.OLT CABS.OLT ε∇	C.ULT CABS.ULT ε∇	C.OLE CABS.OLE ε∇	C.ULE CABS.ULE ε∇
7	111	C.SF CABS.SF ε∇	C.NGLE CABS.NGLE ε∇	C.SEQ CABS.SEQ ε∇	C.NGL CABS.NGL ε∇	C.LT CABS.LT ε∇	C.NGE CABS.NGE ε∇	C.LE CABS.LE ε∇	C.NGT CABS.NGT ε∇

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

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	*	*	*	*	*	*	*
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	CVT.S	CVT.D	*	*	*	*	CVT.PS.PW ε∇	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

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

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

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD ∇	SUB ∇	MUL ∇	*	*	ABS ∇	MOV ∇	NEG ∇
1	001	*	*	*	*	*	*	*	*
2	010	*	MOVCF δ∇	MOVZ ∇	MOVN ∇	*	*	*	*
3	011	ADDR ε∇	*	MULR ε∇	*	RECIP2 ε∇	RECIP1 ε∇	RSQRT1 ε∇	RSQRT2 ε∇
4	100	CVT.S.PU ∇	*	*	*	CVT.PW.PS ε∇	*	*	*
5	101	CVT.S.PL ∇	*	*	*	PLL.PS ∇	PLU.PS ∇	PUL.PS ∇	PUU.PS ∇
6	110	C.F ∇ CABS.F ε∇	C.UN ∇ CABS.UN ε∇	C.EQ ∇ CABS.EQ ε∇	C.UEQ ∇ CABS.UEQ ε∇	C.OLT ∇ CABS.OLT ε∇	C.ULT ∇ CABS.ULT ε∇	C.OLE ∇ CABS.OLE ε∇	C.ULE ∇ CABS.ULE ε∇
7	111	C.SF ∇ CABS.SF ε∇	C.NGLE ∇ CABS.NGLEε∇	C.SEQ ∇ CABS.SEQ ε∇	C.NGL ∇ CABS.NGL ε∇	C.LT ∇ CABS.LT ε∇	C.NGE ∇ CABS.NGE ε∇	C.LE ∇ CABS.LE ε∇	C.NGT ∇ CABS.NGT ε∇

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

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

tf	bit 16	
	0	1
	MOVF.fmt	MOVT.fmt

Table A-22 MIPS64 *COP2* Encoding of rs Field

rs		bits 23..21							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC2 θ	DMFC2 θ⊥	CFC2 θ	MFHC2 θ⊕	MTC2 θ	DMTC2 θ⊥	CTC2 θ	MTHC2 θ⊕
1	01	BC2 θ	*	*	*	*	*	*	*
2	10	*C2* θδ							
3	11								

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

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	LWXC1 ∇	LDXC1 ∇	*	*	*	LUXC1 ∇	*	*
1	001	SWXC1 ∇	SDXC1 ∇	*	*	*	SUXC1 ∇	*	PREFX ∇
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	ALNV.PS ∇	*
4	100	MADD.S ∇	MADD.D ∇	*	*	*	*	MADD.PS ∇	*
5	101	MSUB.S ∇	MSUB.D ∇	*	*	*	*	MSUB.PS ∇	*
6	110	NMADD.S ∇	NMADD.D ∇	*	*	*	*	NMADD.PS ∇	*
7	111	NMSUB.S ∇	NMSUB.D ∇	*	*	*	*	NMSUB.PS ∇	*

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

A.3 Floating Point Unit Instruction Format Encodings

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

Table A-24 Floating Point Unit Instruction Format Encodings

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

Revision History

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

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

Revision	Date	Description
0.90	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: • Correct sign in description of MSUBU. • Update JR and JALR instructions to reflect the changes required by MIPS16.
0.95	March 12, 2001	Update for second external review release.
1.00	August 29, 2002	Updated based on feedback from all reviews. • Add missing optional select field syntax in mtc0/mfc0 instruction descriptions. • Correct the PREF instruction description to acknowledge that the PrepareForStore function does, in fact, modify architectural state. • To provide additional flexibility for Coprocessor 2 implementations, extend the *sel* field for DMFC0, DMTC0, MFC0, and MTC0 to be 8 bits. • Update the PREF instruction to note that it may not update the state of a locked cache line. • Remove obviously incorrect documentation in DIV and DIVU with regard to putting smaller numbers in register *rt*. • Fix the description for MFC2 to reflect data movement from the coprocessor 2 register to the GPR, rather than the other way around. • Correct the pseudo code for LDC1, LDC2, SDC1, and SDC2 for a MIPS32 implementation to show the required word swapping. • Indicate that the operation of the CACHE instruction is UNPREDICTABLE if the cache line containing the instruction is the target of an invalidate or writeback invalidate. • Indicate that an Index Load Tag or Index Store Tag operation of the CACHE instruction must not cause a cache error exception. • Make the entire right half of the MFC2, MTC2, CFC2, CTC2, DMFC2, and DMTC2 instructions implementation dependent, thereby acknowledging that these fields can be used in any way by a Coprocessor 2 implementation. • Clean up the definitions of LL, SC, LLD, and SCD. • Add a warning that software should not use non-zero values of the stype field of the SYNC instruction. • Update the compatibility and subsetting rules to capture the current requirements.

Revision	Date	Description
1.90	September 1, 2002	Merge the MIPS Architecture Release 2 changes in for the first release of a Relesae 2 processor. Changes in this revision include: • All new Release 2 instructions have been included: DEXT, DEXTM, DEXTU, DI, DINS, DINSM, DINSU, DROTR, DROTR32, DROTRV, DSBH, DSHD, EHB, EI, EXT, INS, JALR.HB, JR.HB, MFHC1, MFHC2, MTHC1, MTHC2, RDHWR, RDPGPR, ROTR, ROTRV, SEB, SEH, SYNCI, WRPGPR, WSBH. • The following instruction definitions changed to reflect Release 2 of the Architecture: DERET, DSRL, DSRL32, DSRLV, ERET, JAL, JALR, JR, SRL, SRLV • With support for 64-bit FPUs on 32-bit CPUs in Release 2, all floating point instructions that were previously implemented by MIPS64 processors have been modified to reflect support on either MIPS32 or MIPS64 processors in Release 2. • All pseudo-code functions have been udpated, and the Are64bitFPOperationsEnabled function was added. • Update the instruction encoding tables for Release 2.
2.00	June 9, 2003	Continue with updates to merge Release 2 changes into the document. Changes in this revision include: • Correct the target GPR (from rd to rt) in the SLTI and SLTIU instructions. This appears to be a day-one bug. • Correct CPR number, and missing data movement in the pseudocode for the MTC0 instruction. • Add note to indicate that the CACHE instruction does not take Address Error Exceptions due to mis-aligned effective addresses. • Update SRL, ROTR, SRLV, ROTRV, DSRL, DROTR, DSRLV, DROTRV, DSRL32, and DROTR32 instructions to reflect a 1-bit, rather than a 4-bit decode of shift vs. rotate function. • Add programming note to the PrepareForStore PREF hint to indicate that it can not be used alone to create a bzero-like operation. • Add note to the PREF and PREFX instruction indicating that they may cause Bus Error and Cache Error exceptions, although this is typically limited to systems with high-reliability requirements. • Update the SYNCI instruction to indicate that it should not modify the state of a locked cache line. • Establish specific rules for when multiple TLB matches can be reported (on writes only). This makes software handling easier.

Revision	Date	Description
2.50	July 1, 2005	Changes in this revision: • Correct figure label in LWR instruction (it was incorrectly specified as LWL). • Update all files to FrameMaker 7.1. • Include support for implementation-dependent hardware registers via RDHWR. • Indicate that it is implementation-dependent whether prefetch instructions cause EJTAG data breakpoint exceptions on an address match, and suggest that the preferred implementation is not to cause an exception. • Correct the MIPS32 pseudocode for the LDC1, LDXC1, LUXC1, SDC1, SDXC1, and SUXC1 instructions to reflect the Release 2 ability to have a 64-bit FPU on a 32-bit CPU. The correction simplfies the code by using the ValueFPR and StoreFPR functions, which correctly implement the Release 2 access to the FPRs. • Add an explicit recommendation that all cache operations that require an index be done by converting the index to a kseg0 address before performing the cache operation. • Expand on restrictions on the PREF instruction in cases where the effective address has an uncached coherency attribute.