MIPS
by Imagination

MIPS® Architecture For Programmers
Volume II-A: The MIPS64® Instruction Set

Document Number: MD00087
Revision 6.00
April 5, 2014

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

MIPS Verified™

Template: nB1.03, Built with tags: 2B ARCH FPU_PS FPU_PSandARCH MIPS32 MIPS32andIMPL

Table of Contents

List of Figures

List of Tables

Chapter 1

About This Book

The MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set comes as part of a multi-volume set.

- Volume I-A describes conventions used throughout the document set, and provides an introduction to the MIPS64® Architecture
- Volume I-B describes conventions used throughout the document set, and provides an introduction to the microMIPS64™ Architecture
- Volume II-A provides detailed descriptions of each instruction in the MIPS64® instruction set
- Volume II-B provides detailed descriptions of each instruction in the microMIPS64™ instruction set
- Volume III describes the MIPS64® and microMIPS64™ Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS® processor implementation
- Volume IV-a describes the MIPS16e™ Application-Specific Extension to the MIPS64® Architecture. Beginning with Release 3 of the Architecture, microMIPS is the preferred solution for smaller code size. Release 6 removes MIPS16e: MIPS16e cannot be implemented with Release 6.
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64® Architecture and microMIPS64™. With Release 5 of the Architecture, MDMX is deprecated. MDMX and MSA can not be implemented at the same time. Release 6 removes MDMX: MDMX cannot be implemented with Release 6.
- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS® Architecture. Release 6 removes MIPS-3D: MIPS-3D cannot be implemented with Release 6.
- Volume IV-d describes the SmartMIPS®Application-Specific Extension to the MIPS32® Architecture and the microMIPS32™ Architecture and is not applicable to the MIPS64® document set nor the microMIPS64™ document set. Release 6 removes SmartMIPS: SmartMIPS cannot be implemented with Release 6, neither MIPS32r6 nor MIPS64r6.
- Volume IV-e describes the MIPS® DSP Module to the MIPS® Architecture.
- Volume IV-f describes the MIPS® MT Module to the MIPS® Architecture
- Volume IV-h describes the MIPS® MCU Application-Specific Extension to the MIPS® Architecture
- Volume IV-i describes the MIPS® Virtualization Module to the MIPS® Architecture
- Volume IV-j describes the MIPS® SIMD Architecture Module to the MIPS® Architecture

1.1 Typographical Conventions

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

1.1.1 Italic Text

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

1.1.2 Bold Text

- represents a term that is being **defined**
- is used for **bits** and **fields** that are important from a hardware perspective (for instance, **register** bits, which are not programmable but accessible only to hardware)
- is used for ranges of numbers; the range is indicated by an ellipsis. For instance, **5..1** indicates numbers 5 through 1
- is used to emphasize **UNPREDICTABLE** and **UNDEFINED** behavior, as defined below.

1.1.3 Courier Text

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

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

Table 1.1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
←	Assignment
=, ≠	Tests for equality and inequality
||	Bit string concatenation
x^y	A *y*-bit string formed by *y* copies of the single-bit value *x*
b#n	A constant value *n* in base *b*. For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10.
0bn	A constant value *n* in base *2*. For instance 0b100 represents the binary value 100 (decimal 4).
0xn	A constant value *n* in base *16*. For instance 0x100 represents the hexadecimal value 100 (decimal 256).

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
$x_{y..z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation (rightmost bit is 0) is used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
x.bit[y]	Bit *y* of bitstring *x*. Alternative to the traditional MIPS notation x_y.
x.bits[y..z]	Selection of bits *y* through *z* of bit string *x*. Alternative to the traditional MIPS notation $x_{y..z}$.
x.byte[y]	Byte *y* of bitstring *x*. Equivalent to the traditional MIPS notation $x_{8*y+7..8*y}$.
x.bytes[y..z]	Selection of bytes *y* through *z* of bit string *x*. Alternative to the traditional MIPS notation $x_{8*y+7..8*z}$.
x.halfword[y] x.word[i] x.doubleword[i]	Similar extraction of particular bitfields (used in e.g., MSA packed SIMD vectors).
x.bit31, x.byte0, etc.	Examples of abbreviated form of x.bit[y], etc. notation, when y is a constant.
x.fieldy	Selection of a named subfield of bitstring *x*, typically a register or instruction encoding. More formally described as “Field y of register x”. For example, FIR.D = “the D bit of the Coprocessor 1 Floating-point Implementation Register (FIR)”.
+, −	2’s complement or floating point arithmetic: addition, subtraction
*, ×	2’s complement or floating point multiplication (both used for either)
div	2’s complement integer division
mod	2’s complement modulo
/	Floating point division
<	2’s complement less-than comparison
>	2’s complement greater-than comparison
≤	2’s complement less-than or equal comparison
≥	2’s complement greater-than or equal comparison
nor	Bitwise logical NOR
xor	Bitwise logical XOR
and	Bitwise logical AND
or	Bitwise logical OR
not	Bitwise inversion
&&	Logical (non-Bitwise) AND
<<	Logical Shift left (shift in zeros at right-hand-side)
>>	Logical Shift right (shift in zeros at left-hand-side)
GPRLEN	The length in bits (32 or 64) of the CPU general-purpose registers
GPR[x]	CPU general-purpose register *x*. The content of *GPR[0]* is always zero. In Release 2 of the Architecture, GPR[x] is a short-hand notation for *SGPR[$SRSCtl_{CSS}$, x]*.
SGPR[s,x]	In Release 2 of the Architecture and subsequent releases, multiple copies of the CPU general-purpose registers may be implemented. *SGPR[s,x]* refers to GPR set *s*, register *x*.
FPR[x]	Floating Point operand register *x*
FCC[CC]	Floating Point condition code CC. *FCC[0]* has the same value as *COC[1]*. Release 6 removes the floating point condition codes.
FPR[x]	Floating Point (Coprocessor unit 1), general register *x*

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

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 labeled **I+1** appears to occur "at the same time" as the effect of pseudocode statements labeled **I** for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur "at the same time," there is no defined order. Programs must not depend on a particular order of evaluation between such sections.
PC	The *Program Counter* value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to *PC* during an instruction time. If no value is assigned to *PC* during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the *PC* during the instruction time of the instruction in the branch delay slot. In the MIPS Architecture, the PC value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. Release 6 adds PC-relative address computation and load instructions. The PC value contains a full 64-bit address, all of which are significant during a memory reference.

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
ISA Mode	In processors that implement the MIPS16e Application Specific Extension or the microMIPS base architectures, the *ISA Mode* is a single-bit register that determines in which mode the processor is executing, as follows: **Encoding** / **Meaning** 0 — The processor is executing 32-bit MIPS instructions 1 — The processor is executing MIIPS16e or microMIPS instructions In the MIPS Architecture, the *ISA Mode* value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the *ISA Mode* into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception.
PABITS	The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes.
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 Release 1, the FPU has 32, 32-bit FPRs, in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, (and optionally in MIPS32 Release2 and Release 3) the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR. In MIPS32 Release 1 implementations, **FP32RegistersMode** is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case **FP32RegisterMode** is computed from the FR bit in the *Status* register. If this bit is a 0, the processor operates as if it had 32, 32-bit FPRs. If this bit is a 1, the processor operates with 32 64-bit FPRs. The value of **FP32RegistersMode** is computed from the FR bit in the *Status* register.
InstructionInBranchDelaySlot	Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the *dynamic* state of the instruction, not the *static* state. That is, the value is false if a branch or jump occurs to an instruction whose PC immediately follows a branch or jump, but which is not executed in the delay slot of a branch or jump.
SignalException(exception, argument)	Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument). Control does not return from this pseudocode function—the exception is signaled at the point of the call.

1.4 Notation for Register Field Accessibility

In this document, the read/write properties of register fields use the notations shown in Table 1.1.

Table 1.2 Read/Write Register Field Notation

Read/Write Notation	Hardware Interpretation	Software Interpretation
R/W	A field in which all bits are readable and writable by software and, potentially, by hardware. Hardware updates of this field are visible by software read. Software updates of this field are visible by hardware read. If the Reset State of this field is ‘‘Undefined’’, either software or hardware must initialize the value before the first read will return a predictable value. This should not be confused with the formal definition of **UNDEFINED** behavior.	

Table 1.2 Read/Write Register Field Notation (Continued)

Read/Write Notation	Hardware Interpretation	Software Interpretation
R	A field which is either static or is updated only by hardware. If the Reset State of this field is either ‘‘0’’, ‘‘Preset’’, or ‘‘Externally Set’’, hardware initializes this field to zero or to the appropriate state, respectively, on powerup. The term ‘‘Preset’’ is used to suggest that the processor establishes the appropriate state, whereas the term ‘‘Externally Set’’ is used to suggest that the state is established via an external source (e.g., personality pins or initialization bit stream). These terms are suggestions only, and are not intended to act as a requirement on the implementation. If the Reset State of this field is ‘‘Undefined’’, hardware updates this field only under those conditions specified in the description of the field.	A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware. If the Reset State of this field is ‘‘Undefined’’, software reads of this field result in an **UNPREDICTABLE** value except after a hardware update done under the conditions specified in the description of the field.
R0	R0 = reserved, read as zero, ignore writes by software. Hardware ignores software writes to an R0 field. Neither the occurrence of such writes, nor the values written, affects hardware behavior. Hardware always returns 0 to software reads of R0 fields. The Reset State of an R0 field must always be 0. If software performs an mtc0 instruction which writes a non-zero value to an R0 field, the write to the R0 field will be ignored, but permitted writes to other fields in the register will not be affected.	**Architectural Compatibility:** R0 fields are reserved, and may be used for not-yet-defined purposes in future revisions of the architecture. When writing an R0 field, current software should only write either all 0s, or, preferably, write back the same value that was read from the field. Current software should not assume that the value read from R0 fields is zero, because this may not be true on future hardware. Future revisions of the architecture may redefine an R0 field, but must do so in such a way that software which is unaware of the new definition and either writes zeros or writes back the value it has read from the field will continue to work correctly. Writing back the same value that was read is guaranteed to have no unexpected effects on current or future hardware behavior. (Except for non-atomicity of such read-writes.) Writing zeros to an R0 field may not be preferred because in the future this may interfere with the operation of other software which has been updated for the new field definition.

Table 1.2 Read/Write Register Field Notation (Continued)

Read/Write Notation	Hardware Interpretation	Software Interpretation
0	**Release 6** Release 6 legacy “0” behaves like R0 - read as zero, nonzero writes ignored. Legacy “0” should not be defined for any new control register fields; R0 should be used instead.	
	HW returns 0 when read. HW ignores writes.	Only zero should be written, or, value read from register.
	pre-Release 6 pre-Release 6 legacy “0” - read as zero, nonzero writes UNDEFINED	
	A field which hardware does not update, and for which hardware can assume a zero value.	A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in **UNDEFINED** behavior of the hardware. Software reads of this field return zero as long as all previous software writes are zero. If the Reset State of this field is ‘‘Undefined’’, software must write this field with zero before it is guaranteed to read as zero.
R/W0	Like R/W, except that writes of non-zero to a R/W0 field are ignored. E.g. Status.NMI	
	Hardware may set or clear an R/W0 bit. Hardware ignores software writes of nonzero to an R/W0 field. Neither the occurrence of such writes, nor the values written, affects hardware behavior. Software writes of 0 to an R/W0 field may have an effect. Hardware may return 0 or nonzero to software reads of an R/W0 bit. If software performs an mtc0 instruction which writes a non-zero value to an R/W0 field, the write to the R/W0 field will be ignored, but permitted writes to other fields in the register will not be affected.	Software can only clear an R/W0 bit. Software writes 0 to an R/W0 field to clear the field. Software writes nonzero to an R/W0 bit in order to guarantee that the bit is not affected by the write.

1.5 For More Information

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

For comments or questions on the MIPS64® Architecture or this document, send Email to IMGBA-DocFeedback@imgtec.com.

Chapter 2

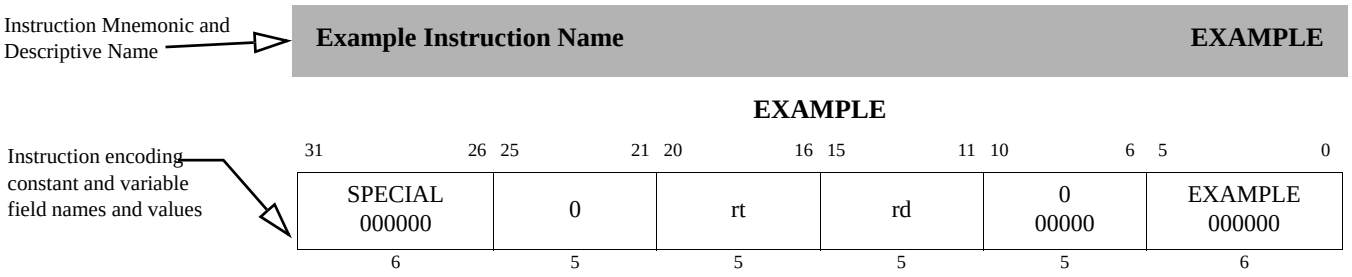
Guide to the Instruction Set

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

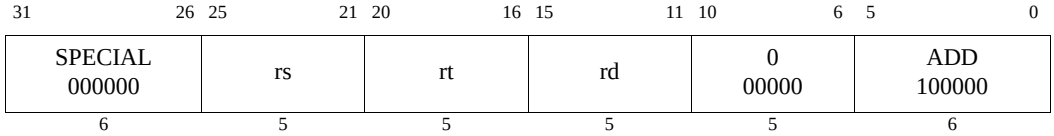


2.1.1 Instruction Fields

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.

Figure 2.3 Example of Instruction Descriptive Name and Mnemonic

Add Word **ADD**

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.

Figure 2.4 Example of Instruction Format

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

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. Instructions introduced at different times by different ISA family members, are indicated by markings such as “MIPS64, MIPS32 Release 2”. Instructions removed by particular architecture release are indicated in the Availability section.

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.

Figure 2.5 Example of Instruction Purpose

Purpose: Add Word

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

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.

Figure 2.6 Example of Instruction Description

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*.

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)

Figure 2.7 Example of Instruction Restrictions

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`.

2.1.7 Availability and Compatibility Fields

The *Availability* and *Compatibility* sections are not provided for all instructions. These sections list considerations relevant to whether and how an implementation may implement some instructions, when software may use such instructions, and how software can determine if an instruction or feature is present. Such considerations include:

- Some instructions are not present on all architecture releases. Sometimes the implementation is required to signal a Reserved Instruction exception, but sometimes executing such an instruction encoding is architecturally defined to give UNPREDICTABLE results.
- Some instructions are available for implementations of a particular architecture release, but may be provided only if an optional feature is implemented. Control register bits typically allow software to determine if the feature is present.
- Some instructions may not behave the same way on all implementations. Typically this involves behavior that was UNPREDICTABLE in some implementations, but which is made architectural and guaranteed consistent so that software can rely on it in subsequent architecture releases.
- Some instructions are prohibited for certain architecture releases and/or optional feature combinations.
- Some instructions may be removed for certain architecture releases. Implementations may then be required to signal a Reserved Instruction exception for the removed instruction encoding; but sometimes the instruction encoding is reused for other instructions.

All of these considerations may apply to the same instruction. If such considerations applicable to an instruction are simple, the architecture level in which an instruction was defined or redefined in the *Format* field, and/or the *Restrictions* section, may be sufficient; but if the set of such considerations applicable to an instruction is complicated, the *Availability* and *Compatibility* sections may be provided.

2.1.8 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.

Figure 2.8 Example of Instruction Operation

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
```

See 2.2 “Operation Section Notation and Functions” on page 29 for more information on the formal notation used here.

2.1.9 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.

Figure 2.9 Example of Instruction Exception

Exceptions:

Integer Overflow

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

2.1.10 Programming Notes

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

Figure 2.10 Example of Instruction Programming Notes

Programming Notes:

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

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.

2.2.2.1.1 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*.

Figure 2.11 COP_LW Pseudocode Function

```
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
```

2.2.2.1.2 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*.

Figure 2.12 COP_LD Pseudocode Function

```
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
```

2.2.2.1.3 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*.

Figure 2.13 COP_SW Pseudocode Function

```
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
```

2.2.2.1.4 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*.

Figure 2.14 COP_SD Pseudocode Function

```
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
```

2.2.2.1.5 CoprocessorOperation

The CoprocessorOperation function performs the specified Coprocessor operation.

Figure 2.15 CoprocessorOperation Pseudocode Function

```
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
```

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.

2.2.2.2.1 Misaligned Support

MIPS processors originally required all memory accesses to be naturally aligned. MSA (the MIPS SIMD Architecture) supported misaligned memory accesses for its 128 bit packed SIMD vector loads and stores, from its introduction in MIPS Release 5. MIPS Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

The pseudocode function MisalignedSupport encapsulates the version number check to determine if mislaignment is supported for an ordinary memory access.

Figure 2.16 MisalignedSupport Pseudocode Function

```
predicate ← MisalignedSupport ()
   return Config.AR ≥ 2 // Architecture Revision 2 corresponds to MIPS Release 6.
end function
```

See Appendix B, “Misaligned Memory Accesses” on page 768 for a more detailed discussion of misalignment, including pseudocode functions for the actual misaligned memory access.

2.2.2.2.2 AddressTranslation

The AddressTranslation function translates a virtual address to a physical address and its cacheability and coherency attribute, 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 cacheability and coherency attribute (*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.

Figure 2.17 AddressTranslation Pseudocode Function

```
(pAddr, CCA) ←AddressTranslation (vAddr, IorD, LorS)
   vAddr ← memory_address(vAddr) /* mode dependent address space wrapping */

   /* pAddr: physical address */
   /* CCA:   Cacheability&Coherency Attribute,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
```

2.2.2.2.3 LoadMemory

The LoadMemory function loads a value from memory.

This action uses cache and main memory as specified in both the Cacheability and Coherency Attribute (*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.

Figure 2.18 LoadMemory Pseudocode Function

```
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:      Cacheability&CoherencyAttribute=method used to access caches */
    /*           and memory and resolve the reference */

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

endfunction LoadMemory
```

2.2.2.2.4 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 Cacheability and Coherency Attribute (*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.

Figure 2.19 StoreMemory Pseudocode Function

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

    /* CCA:      Cacheability&Coherency Attribute, 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
```

2.2.2.2.5 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.

Figure 2.20 Prefetch Pseudocode Function

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

    /* CCA:    Cacheability&Coherency Attribute, 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
```

Table 2.1 lists the data access lengths and their labels for loads and stores.

Table 2.1 AccessLength Specifications for Loads/Stores

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

2.2.2.2.6 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.

Figure 2.21 SyncOperation Pseudocode Function

```
SyncOperation(stype)

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

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

endfunction SyncOperation
```

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).

2.2.2.3.1 ValueFPR

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

Figure 2.22 ValueFPR Pseudocode Function

```
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 (fpr_0 ≠ 0) then
               valueFPR ← UNPREDICTABLE
            else
               valueFPR ← FPR[fpr+1]_31..0 || FPR[fpr]_31..0
            endif
         else
            valueFPR ← FPR[fpr]
         endif

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

```
        DEFAULT:
            valueFPR ← UNPREDICTABLE

    endcase
endfunction ValueFPR
```

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.

2.2.2.3.2 StoreFPR

Figure 2.23 StoreFPR Pseudocode Function

```
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
```

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

2.2.2.3.3 CheckFPException

Figure 2.24 CheckFPException Pseudocode Function

```
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
```

2.2.2.3.4 FPConditionCode

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

Figure 2.25 FPConditionCode Pseudocode Function

```
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
```

2.2.2.3.5 SetFPConditionCode

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

Figure 2.26 SetFPConditionCode Pseudocode Function

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

endfunction SetFPConditionCode
```

such operations are never enabled and this function returns 0`Are64BitFPOperationsEnabled ← 0`

2.2.2.4 Pseudocode Functions Related to Sign and Zero Extension

2.2.2.4.1 Sign extension and zero extension in pseudocode

Much pseudocode uses a generic function `sign_extend` without specifying from what bit position the extension is done, when the intention is obvious. E.g. `sign_extend(immediate16)` or `sign_extend(disp9)`.

However, sometimes it is necessary to specify the bit position. For example, $\texttt{sign\_extend(temp}_{31..0}\texttt{)}$ or the more complicated $\texttt{(offset}_{15}\texttt{)}^{\texttt{GPRLEN-(16+2)}}\ \texttt{||}\ \texttt{offset}\ \texttt{||}\ 0^2$.

The explicit notation `sign_extend.nbits(val)` or `sign_extend(val,nbits)` is suggested as a simplification. They say to sign extend as if an nbits-sized signed integer. The width to be sign extended to is usually apparent by context, and is usually GPRLEN, 32 or 64 bits. The previous examples then become.

$\texttt{sign\_extend(temp}_{31..0}\texttt{)}$
`= sign_extend.32(temp)`

and

$\texttt{(offset}_{15}\texttt{)}^{\texttt{GPRLEN-(16+2)}}\ \texttt{||}\ \texttt{offset}\ \texttt{||}\ 0^2$
`= sign_extend.16(offset)<<2`

Note that sign_extend.N(value) extends from bit position N-1, if the bits are numbered 0..N-1 as is typical.

The explicit notations `sign_extend.nbits(val)` or `sign_extend(val,nbits)` is used as a simplification. These notations say to sign extend as if an nbits-sized signed integer. The width to be sign extended to is usually apparent by context, and is usually GPRLEN, 32 or 64 bits.

Figure 2.27 sign_extend Pseudocode Functions

```
sign_extend.nbits(val) = sign_extend(val,nbits) /* syntactic equivalents */

function sign_extend(val,nbits)
   return (val_nbits-1)^GPRLEN-nbits || val_nbits-1..0
end function
```

The earlier examples can be expressed as

$\texttt{(offset}_{15}\texttt{)}^{\texttt{GPRLEN-(16+2)}}\ \texttt{||}\ \texttt{offset}\ \texttt{||}\ 0^2$
`= sign_extend.16(offset) << 2)`

and

$\texttt{sign\_extend(temp}_{31..0}\texttt{)}$
`= sign_extend.32(temp)`

Similarly for `zero_extension`, although zero extension is less common than sign extension in the MIPS ISA.

Floating point may use notations such as `zero_extend.fmt` corresponding to the format of the FPU instruction. E.g. `zero_extend.S` and `zero_extend.D` are equivalent to `zero_extend.32` and `zero_extend.64`.

Existing pseudocode may use any of these, or other, notations. TBD: rewrite pseudocode.

2.2.2.4.2 Testing sign extension or zero extension

Special provision is made for the inputs to unsigned 32-bit multiplies on a 64-bit CPU. Since many instructions produce sign extend 32 bits to 64 even for unsigned computation, properly sign extended numbers must be accepted as

input, and truncated to 32 bits, clearing bits 32-63. However, it is also desirable to accept zero extended 32-bit integers, with bits 32-63 all 0.[1]

Pre-Release 6 manuals contained the function NotWordValue, which did not accept zero-extended 32 bit values. This is inherited.

The pseudocode function `zero_or_sign_extended.32(value)` permits both sign extended or zero extended values.

Figure 2.28 is_zero_or_sign_extended Pseudocode Functions

```
function is_zero_or_sign_extended.32(val)
   if value63..32 = (value31)32 then return true
   if value63..32 = (0)32 then return true
   return false
end function
```

2.2.2.4.3 memory_address

The pseudocode function `memory_address` performs mode-dependent address space wrapping for compatibility between MIPS32 and MIPS64. It is applied to all memory references. It may be specified explicitly in some places, particularly for new memory reference instructions, but it is also declared to apply implicitly to all memory references as defined below. In addition, certain instructions that are used to calculate effective memory addresses but which are not themselves memory accesses specify memory_address explicitly in their pseudocode.

Figure 2.29 memory_address Pseudocode Function

```
function memory_address(ea)
   if User mode and Status.UX = 0 then return sign_extend.32(ea)
   /* Preliminary proposal to wrap privileged mode addresses */
   if Supervisormode and Status.SX = 0 then return sign_extend.32(ea)
   if Kernel mode and Status.KX = 0 then return sign_extend.32(ea)
   /* if Hardware Page Table Walking, then wrap in same way as Kernel/VZ Root */
   return ea
end function
```

On a 32-bit CPU, `memory_address` returns its 32-bit effective address argument unaffected.

On a 64-bit processor, `memory_address` optionally truncates a 32-bit address by sign extension, It discards carries that may have propagated from the lower 32-bits to the upper 32-bits that would cause minor differences between MIPS32 and MIPS64 execution.It is used in certain modes[2] on a MIPS64 CPU where strict compatibility with MIPS32 is required. This behavior was and continues to be described in a section of Volume III of the MIPS ARM[3]- However, the behavior was not formally described in pseudocode functions prior to Release 6.

In addition to the use of `memory_address` for all memory references (including load and store instructions, LL/SC), Release 6 extends this behavior to control transfers (branch and call instructions), and to the PC-relative address calculation instructions (ADDIUPC, AUIPC, ALUIPC). In newer instructions the function is explicit in the pseudocode.

1. Requiring that both zero or sign extended integers be accepted may complicate the multiplier.
2. Currently, if in User/Supervisor/Kernel mode and Status.UX/SX/KX=0.
3. E.g. see section named “Special Behavior for Data References in User Mode with $Status_{UX}$=0”, in the MIPS(r) Architecture Reference Manual Volume III, the MIPS64(R) and microMIPS64(tm) Privileged Resource Architecture, e.g. in section 4.11 of revision 5.03, or section 4.9 of revision 1.00.

Implicit address space wrapping for all instruction fetches is described by the following pseudocode fragment which should be considered part of instruction fetch:

Figure 2.30 Instruction Fetch Implicit memory_address Wrapping

```
PC ← memory_address( PC )
( instruction_data, length ) ← instruction_fetch( PC )
/* decode and execute instruction */
```

Implicit address space wrapping for all data memory accesses is described by the following pseudocode, which is inserted at the top of the AddressTranslation pseudocode function:

Figure 2.31 AddressTranslation implicit memory_address Wrapping

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

In addition to its use in instruction pseudocode,

2.2.2.5 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

2.2.2.5.1 areAnyBitsSet

Pseudocode function areAnyBitsSet.fmt(fval) is used to emphasize, first that this true/false NEZ any1/all0 testing is being done, second to emphasize that it tests all bits, and is not a floating point numeric comparison, and finally to indicate that the number of bits tested corresponds to the size of the format.

Figure 2.32 areAnyBitsSet Pseudocode Functions

```
function areAnyBitsSet.fmt(val)
   if all bits 0 .. nbits(fmt) are 0
      then return false
      else return true
end function
```


2.2.2.5.2 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.

Figure 2.33 SignalException Pseudocode Function

```
SignalException(Exception, argument)

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

endfunction SignalException
```

2.2.2.5.3 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.

Figure 2.34 SignalDebugBreakpointException Pseudocode Function

```
SignalDebugBreakpointException()

endfunction SignalDebugBreakpointException
```

2.2.2.5.4 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.

Figure 2.35 SignalDebugModeBreakpointException Pseudocode Function

```
SignalDebugModeBreakpointException()

endfunction SignalDebugModeBreakpointException
```

2.2.2.5.5 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.

Figure 2.36 NullifyCurrentInstruction PseudoCode Function

```
NullifyCurrentInstruction()

endfunction NullifyCurrentInstruction
```

2.2.2.5.6 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.

Figure 2.37 JumpDelaySlot Pseudocode Function

```
JumpDelaySlot(vAddr)

   /* vAddr:Virtual address */

endfunction JumpDelaySlot
```

2.2.2.5.7 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.

Figure 2.38 NotWordValue Pseudocode Function

```
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
```

2.2.2.5.8 PolyMult

The PolyMult function multiplies two binary polynomial coefficients.

Figure 2.39 PolyMult Pseudocode Function

```
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
```

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 “Op and Function Subfield Notation” on page 42 for a description of the *op* and *function* subfields.

Chapter 3

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 provides 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* and/or *COP2* major opcodes, by adding control for co-processors via the *COP2*, *LWC2*, *SWC2*, *LDC2*, and/or *SDC2*, or via the addition of approved Application Specific Extensions.MIPSr6 removes all instructions under the SPECIAL2 major opcode, either by removing them or moving them to the COP2 major opcode. Similarly, all coprocessor 2 support instructions (e.g. LWC2) have been moved to the COP2 major opcode. Supersetting of the MIPSr6 architecture is only allowed in the COP2 major opcode, or via the addition of approved Application Specific Extensions. SPECIAL2 is reserved for MIPS.

Note: The use of COP3 as a customizable coprocessor has been removed in the Release 2 of the MIPS64 architecture. The use of the COP3 is now reserved for the future extension of the architecture. The instruction set subsetting rules are described in the subsections below, and also the following rule:

- **Co-dependence of Architecture Features:** MIPSr5™ (also called Release 5) and subsequent releases (such as MIPSr6) include a number of features. Some are optional; some are required. Features provided by a release, such as MIPSr5 or later, whether optional or required, must be consistent. If any feature that is introduced by a particular release is implemented, e.g. which is described as part of Release 5 and not any earlier release, then all other features must be implemented in a manner consistent with that release. For example: the VZ and MSA features are introduced by Release 5 but are optional, whereas the FR=1 64-bit FPU register model was optional when introduced earlier, but is now required by Release 5 if any FPU is implemented. If any or all of VZ or MSA are implemented, then Release 5 is implied, and then if an FPU is implemented, it must implement the FR=1 64-bit FPU register model.

3.1.1 Subsetting of Non-Privileged Architecture

- All non-privileged (do not need access to Coprocessor 0) CPU (non-FPU) instructions must be implemented - no subsetting of these are allowed - per the MIPS Instruction Set Architecture release supported.
- If any instruction is subsetted out based on the rules below, an attempt to execute that instruction must cause the appropriate exception (typically Reserved Instruction or Coprocessor Unusable).
- The FPU and related support instructions, such as CPU conditional branches on FPU conditions (e.g. pre-MIPSr6 BC1T/BC1F, MIPSr6 BC1NEQZ) and CPU conditional moves on FPU conditions (e.g. pre-MIPSr6 MOVT/MOVF), may be omitted. Software may determine if an FPU is implemented by checking the state of the FP bit in the *Config1* CP0 register. Software may determine which FPU data types are implemented by checking the appropriate bits in the *FIR* CP1 register. The following allowable FPU subsets are compliant with the MIPS64 architecture:
  - No FPU

Config1.FP=0

- FPU with S, and W formats and all supporting instructions.

  This 32-bit subset is permitted by MIPSr6, but prohibited by pre-MIPSr6 releases.

  Config1.FP=1, Status.FR=0, FIR.S=FIR.L=1, FIR.D=FIR.L=FIR.PS=0.

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

  Config1.FP=1, Status.FR=(see below), FIR.S=FIR.L=FIR.D=FIR.L=1, FIR.PS=0.

  pre-MIPSr5 permits this 64-bit configuration, and allows both FPU register modes. Status.FR=0 support is required but Status.FR=1 support is optional.

  MIPSr5 permits this 64-bit configuration, and requires both FPU register modes, i.e. both Status.FR=0 and Status.FR=1 support are required.

  MIPSr6 permits this 64-bit configuration, but requires Status.FR=1 and FIR.F64=1. MIPSr6 prohibits Status.FR=0 if FIR.D=1 or FIR.L=1.

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

  Config1.FP=1, Status.FR=0/1, FIR.S=FIR.L=FIR.D=FIR.L=FIR.PS=1.

  MIPSr6 prohibits this mode, and any mode with FIR.PS=1 paired single support.

- In Release 5 of the Architecture, if floating point is implemented then FR=1 is required. I.e. the 64-bit FPU, with the FR=1 64-bit FPU register model, is required. The FR=0 32-bit FPU register model continues to be required.

- 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.

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

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

- Supported Modules/ASEs are optional and may be subsetted out. If most cases, software may determine if a supported Module/ASE is implemented by checking the appropriate bit in the *Config1* or *Config3* or *Config4* CP0 register. If they are implemented, they must implement the entire ISA applicable to the component, or implement subsets that are approved by the Module/ASE specifications.

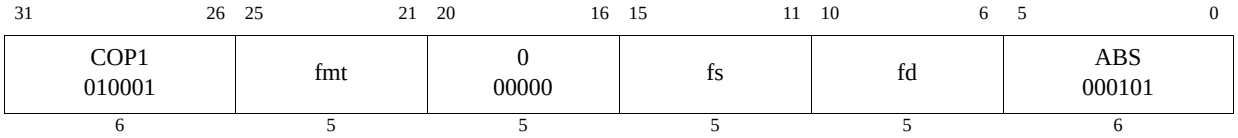
- EJTAG is optional and may be subsetted out. If it is implemented, it must implement only those subsets that are approved by the EJTAG specification. If EJTAG is not implemented, the EJTAG instructions (SDBBP and DERET) can be subsetted out.

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

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

3.2 Alphabetical List of Instructions

This section contains individual instruction descriptions, arranged in alphabetical order.



Format: `ABS.fmt`

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

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

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

Purpose: Floating Point Absolute 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.

If $FIR_{Has2008}$=0 or $FCSR_{ABS2008}$=0 then this operation is arithmetic. For this case, any NaN operand signals invalid operation.

If $FCSR_{ABS2008}$=1 then this operation is non-arithmetic. For this case, both regular floating point numbers and NAN values are treated alike, only the sign bit is affected by this instruction. No IEEE exception can be generated for this case.

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 the *FR*=0 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

The ABS.PS instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

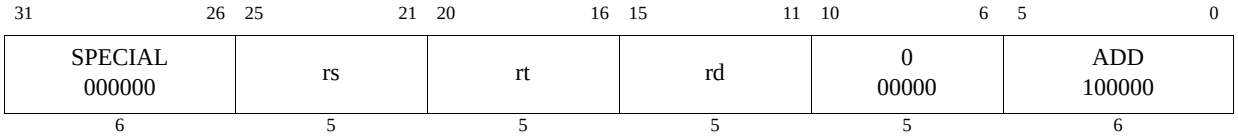
```
StoreFPR(fd, fmt, AbsoluteValue(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation



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

Purpose: Add Word

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:

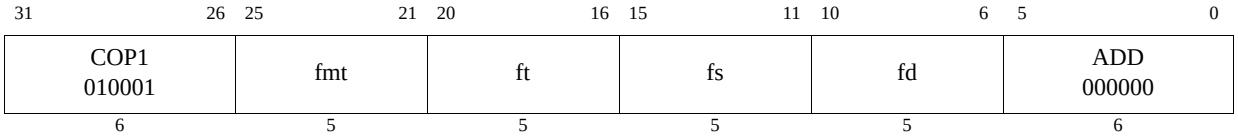
```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← sign_extend(temp31..0)
endif
```

Exceptions:

Integer Overflow

Programming Notes:

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



Format: `ADD.fmt`

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

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

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

Purpose: Floating Point Add

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 the *FR*=0 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

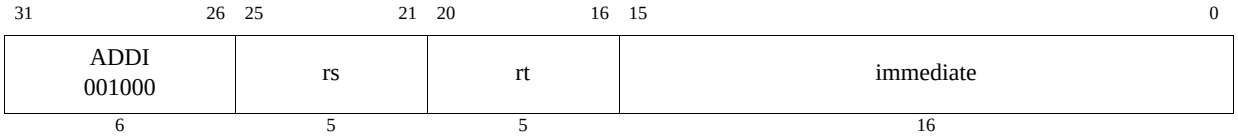
```
StoreFPR (fd, fmt, ValueFPR(fs, fmt) +fmt ValueFPR(ft, fmt))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow



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

Purpose: Add Immediate Word

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**.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture. The encoding has been reused for other instructions introduced by MIPS32 Release 6.

Operation:

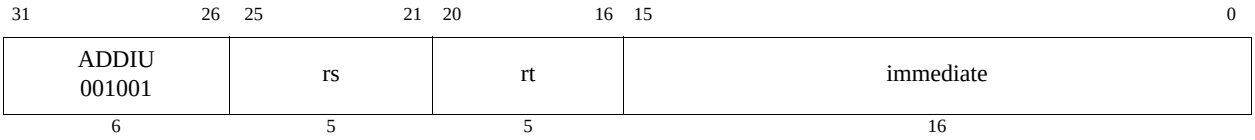
```
if NotWordValue(GPR[rs]) then
   UNPREDICTABLE
endif
temp ← (GPR[rs]31||GPR[rs]31..0) + sign_extend(immediate)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rt] ← sign_extend(temp31..0)
endif
```

Exceptions:

Integer Overflow

Programming Notes:

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



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

Purpose: Add Immediate Unsigned Word

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:

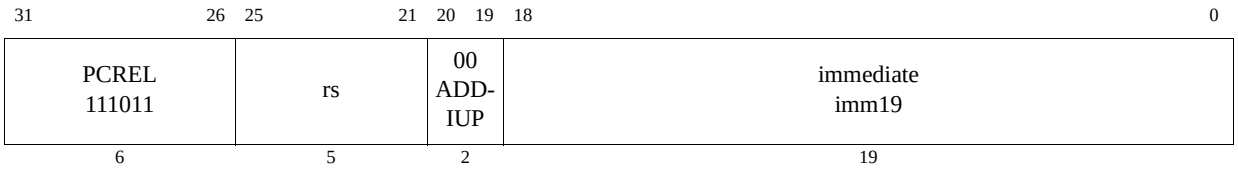
```
if NotWordValue(GPR[rs]) then
   UNPREDICTABLE
endif
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt] ← sign_extend(temp31..0)
```

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.



Format: `ADDIUPC`
`ADDIUPC rs,imm19` **MIPS32 Release 6**

Purpose: Add Immediate to PC (unsigned - non-trapping)[1]

Description: `GPR[rs] ← memory_address( PC + sign_extend( imm19 << 2 ) )`

The 19 bit immediate is shifted left by 2 bits, sign-extended, and added to the address of the ADDIUPC instruction.

The result of the addition is placed in GPR `rs`.

Restrictions:

Availability:

ADDIUPC is introduced by and required as of MIPS32 Release 6.

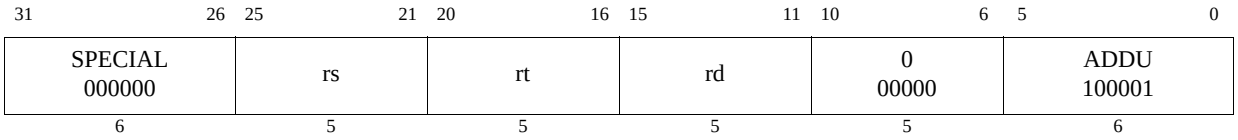
Operation

```
GPR[rd] ← memory_address( PC + sign_extend( imm19 << 2 ) )
```

Exceptions:

None.

1. The term “unsigned” in this instruction mnemonic is a misnomer. “Unsigned” here means “non-trapping”. It does not trap on a signed 32-bit overflow. ADDIUPC corresponds to unsigned ADDIU which does not trap on overflow, rather than ADDI, which traps on overflow.



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

Purpose: Add Unsigned Word

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:

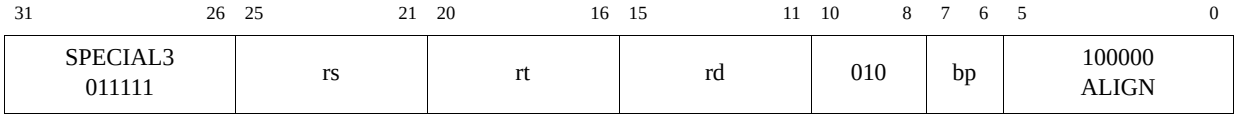
```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← sign_extend(temp31..0)
```

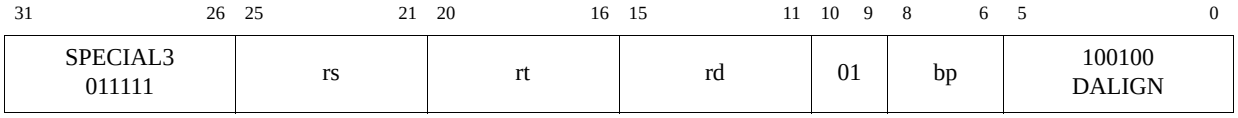
Exceptions:

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: `ALIGN DALIGN`
`ALIGN rd,rs,rt,bp` **MIPS32 Release 6**
`DALIGN rd,rs,rt,bp` **MIPS64 Release 6**

Purpose: Concatenate two GPRs, and extract a contiguous subset at a byte position

Description: `GPR[rd] ← concatenate(GPR[rt],GPR[rs]) >> (GPRLEN-8*bp)))`

Or equivalently

`GPR[rd] ← (GPR[rt] << (8*bp)) || (GPR[rs] >> (GPRLEN-8*bp))`

Conceptually, the input registers GPR `rt` and GPR `rs` are concatenated, and a register width contiguous subset is extracted, specified by the byte pointer `bp`.

The ALIGN instruction operates on 32-bit words, and has a 2-bit byte position field `bp`.

The DALIGN instruction operates on 64-bit doublewords, and has a 3-bit byte position field `bp`.

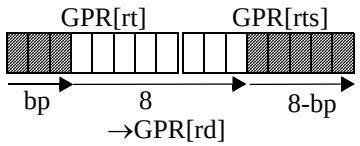
In detail:

ALIGN: The rightmost 32-bit word in GPR `rt` is left shifted as a 32-bit value by `bp` byte positions. The rightmost 32-bit word in register `rs` is right shifted as a 32-bit value by `(4-bp)` byte positions. These shifts are logical shifts, zero-filling. The shifted values are then *or*-ed together to create a 32-bit result that is sign-extended to 64-bits and written to destination GPR `rd`.

DALIGN: The 64-bit doubleword in GPR `rt` is left shifted as a 64 bit value by `bp` byte positions. The 64-bit word in register `rs` is right shifted as a 64-bit value by `(8-bp)` byte positions. These shifts are logical shifts, zero-filling. The shifted values are then *or*-ed together to create a 64-bit result and written to destination GPR `rd`.

Graphically:

Figure 3.1 DALIGN operation (64-bit)



Restrictions:

Executing ALIGN and DALIGN with shift count `bp=0` acts like a register to register move operation, is redundant with other such, and is therefore discouraged. Software should not generate ALIGN or DALIGN with shift count

bp=0.

Availability:

The ALIGN instruction is introduced by and required as of MIPS32 Release 6.

The DALIGN instruction is introduced by and required as of MIPS64 Release 6.

Operation

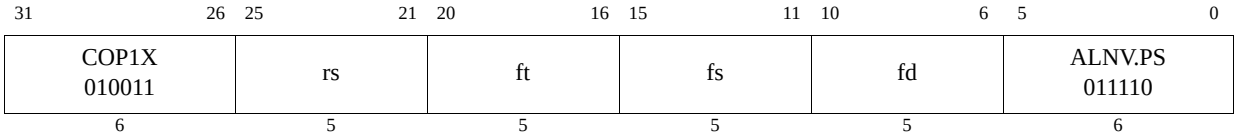
```
ALIGN:
  tmp_rt_hi ← unsigned_word(GPR[rt]) << (8*bp)
  tmp_rs_lo ← unsigned_word(GPR[rs]) >> (8*(4-bp))
  tmp ← tmp_rt_hi || tmp_rt_lo
  GPR[rd] ← sign_extend.32(tmp)



  tmp_rt_hi ← unsigned_doubleword(GPR[rt]) << (8*bp)
  tmp_rs_lo ← unsigned_doubleword(GPR[rs]) >> (8*(8-bp))
  tmp ← tmp_rt_hi || tmp_rt_lo
  GPR[rd] ← tmp
```

Exceptions:

None

Programming Notes:

The Release 6 ALIGN instruction corresponds to the pre-Release 6 DSP Module BALIGN instruction, except that BALIGN with shift counts of 0 and 2 are specified as being UNPREDICTABLE, whereas ALIGN and DALIGN define all bp values, discouraging only bp=0.



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

Purpose: Floating Point Align Variable

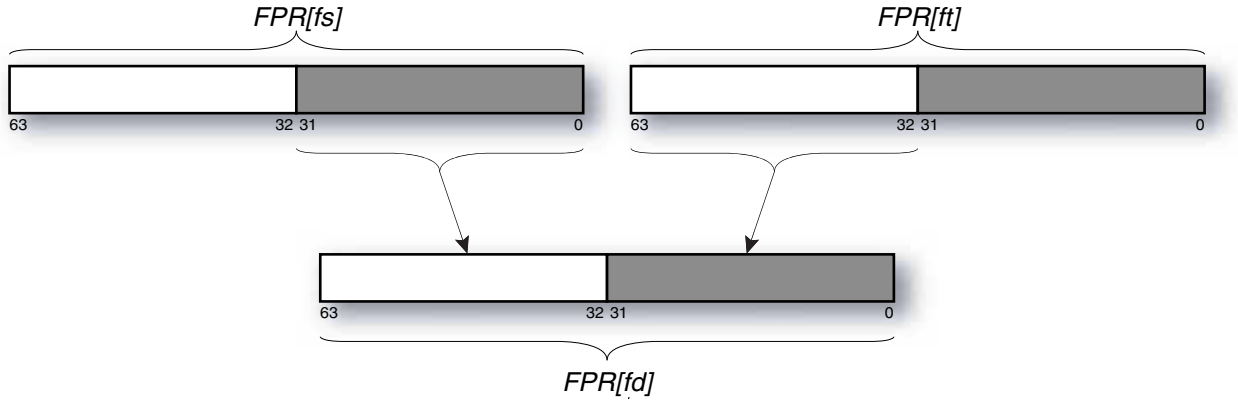
To align a misaligned pair of paired single values

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 3.2 Example of an ALNV.PS Operation



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

Restrictions:

This instruction has been removed[1] in the MIPS32 Release 6 architecture.

Prior to MIPS32 Release 6, the following restrictions apply:

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 the *FR*=0 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

1. "Removed by MIPS32 Release 6" means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.

Availability:

This instruction has been removed in the Release 6 architecture.

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:

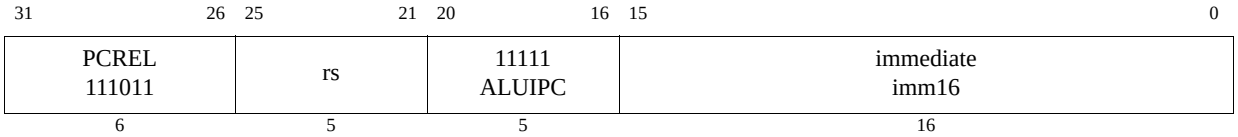
ALNV.PS is designed to be used with LUXC1 to load 8 bytes of data from any 4-byte boundary. For example:

```
/* 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: `ALUIPC`
`ALUIPC rs,imm16` **MIPS32 Release 6**

Purpose: Aligned Add Upper Immediate to PC

Description: `~0x0FFFF & memory_address( PC + sign_extend( imm16 << 16 ) )`

The 16 bit immediate is shifted left 16 bits, sign-extended, and added to the address of the ALUIPC instruction.

The low 16 bits of the result are cleared; i.e. the address is aligned on a 64K boundary.

The result of the addition is placed in GPR `rs`.

Restrictions:

Availability:

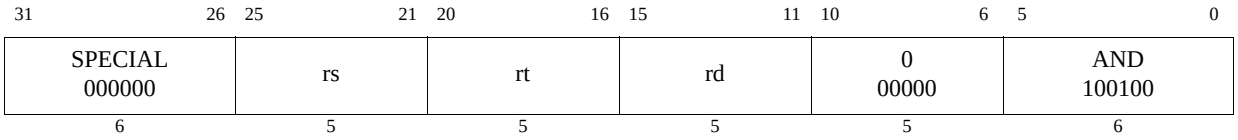
ALUIPC introduced by and required as of MIPS32 Release 6.

Operation

```
GPR[rs] ← ~0x0FFFF & memory_address( PC + sign_extend( imm16 << 16 ) )
```

Exceptions:

None.



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

Purpose: And

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:

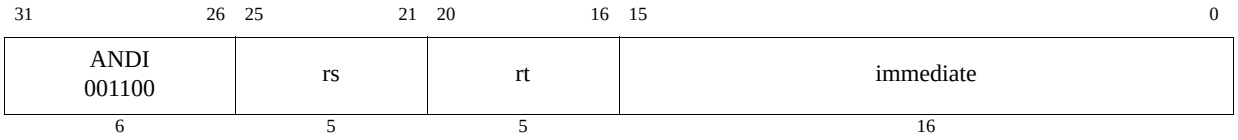
None

Operation:

```
GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None



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

Purpose: And Immediate

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:

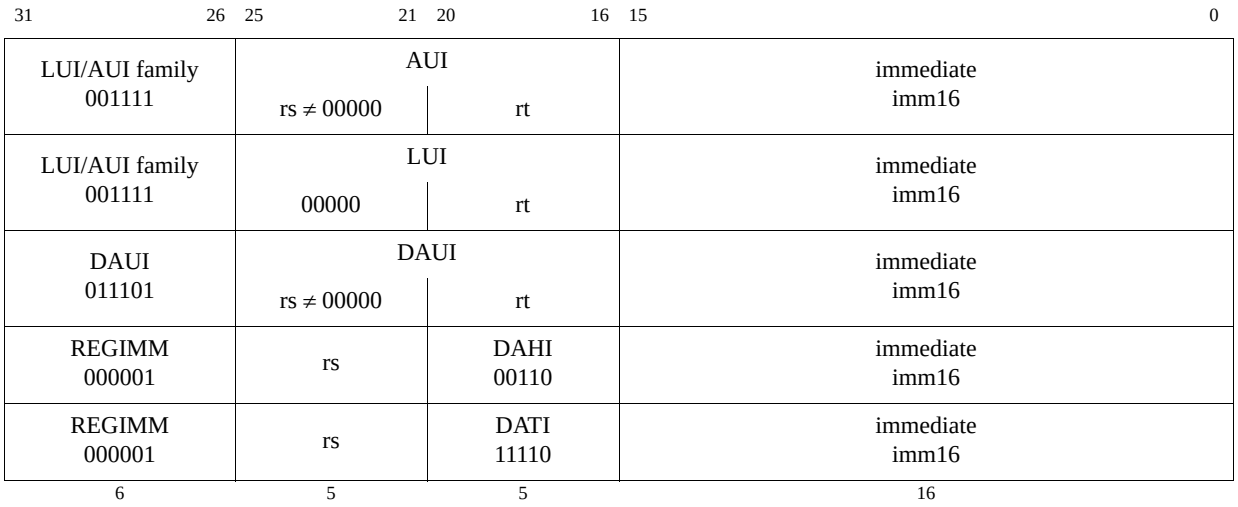
None

Operation:

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

Exceptions:

None



31 – 26	25 – 21	20 – 16	15 – 0
LUI/AUI family 001111	AUI: rs ≠ 00000	rt	immediate imm16
LUI/AUI family 001111	LUI: 00000	rt	immediate imm16
DAUI 011101	DAUI: rs ≠ 00000	rt	immediate imm16
REGIMM 000001	rs	DAHI 00110	immediate imm16
REGIMM 000001	rs	DATI 11110	immediate imm16
6	5	5	16

Format: `AUI LUI DAUI DAHI DATI`

`AUI`

```
AUI rs, rt, imm16
```

MIPS32 Release 6

`LUI`

```
LUI rt, imm16
```

MIPS32 Release 6

`DAUI`

```
DAUI rs, rt, imm16
```

MIPS64 Release 6

`DAHI`

```
DAHI rs, imm16
```

MIPS64 Release 6

`DATI`

```
DATI rs, imm16
```

MIPS64 Release 6

Purpose: Add/Load Immediate to Upper Bits

AUI: Add Upper Immediate
LUI: Load Upper Immediate (Assembly Idiom in Release 6)

DAUI: Doubleword Add Upper Immediate
DLUI: Doubleword Load Upper Immediate (Assembly Idiom in Release 6)

DAHI: Doubleword Add Higher Immediate

DATI: Doubleword Add Top Immediate

Description:

```
AUI:  rt ← sign_extend.32( rs + sign_extend(imm16 << 16) )

DAUI:  rt ← rs + sign_extend(imm16 << 16)
DAHI: rs ← rs + sign_extend(imm16 << 32)
```

```
DATI: rs ← rs + sign_extend(imm16 << 48)

LUI:  rt ← sign_extend.32( imm16 << 16)
```

AUI: The 16 bit immediate is shifted left 16 bits, sign-extended, and added to the register `rs`, storing the result in `rt`. AUI is a 32-bit compatible instruction, so on a 64-bit CPU the result is sign extended as if a 32-bits signed address.

LUI The 16 bit immediate is shifted left 16 bits and sign-extended. LUI may be considered a special case of MIPS32 Release 6 AUI with rs=00000.

DAHI: The 16 bit immediate is shifted left 32 bits, sign-extended, and added to the register `rs`, overwriting `rs` with the result.

DATI: The 16 bit immediate is shifted left 48bits, sign-extended, and added to the register `rs`, overwriting `rs` with the result.

LUI is a pre-Release 6 instruction that may be considered to be an assembly code idiom for AUI with rs=00000 in MIPS32 Release 6. 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*.

DLUI: The 16-bit *immediate* is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is anti-sign-extended and placed into GPR *rt*. “Anti-sign-extended” means that the inverse of the sign bit, bit 31, is extended to bits 32 to 63.

Equivalently: 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 then XORed with 0xFFFF.FFFF.0000.0000 or ~0x0FFFFFFF to invert the sign extension.

Restrictions:

Availability:

AUI is introduced by and required as of MIPS32 Release 6.

LUI may be considered a special case of MIPS32 Release 6 AUI with rs=00000, but LUI has existed in all releases of the MIPS ISA.

DAUI is introduced by and required as of MIPS64 Release 6.

DLUI is introduced by and required as of MIPS64 Release 6.

DAUI and DLUI reuse the primary opcode 011101 JALX: DAUI and DLUI cannot be trapped (and possibly emulated) on pre-Release 6 systems, whiled JALX cannot be trapped (and possibly emulated) on MIPS32 Release 6 systems.

DAHI is introduced by and required as of MIPS64r6.

DATI is introduced by and required as of MIPS64r6.

Operation

```
AUI:  GPR[rt] ← sign_extend.32( GPR[rs] + sign_extend(imm16 << 16) )
DAHI: GPR[rs] ← GPR[rs] + sign_extend(imm16 << 32)
DATI: GPR[rs] ← GPR[rs] + sign_extend(imm16 << 48)
```

Exceptions:

AUI, LUI: None

DAUI, DLUI, DAHI, DATI: Reserved Instruction Exception if 64-bit instructions are not enabled.

Programming Notes:

AUI (and DAUI, DLUI, DAHI and DATI on MIPS64r6) can be used to synthesize large constants in situations where

it is not convenient to load a large constant from memory. To simplify hardware that may recognize sequences of instructions as generating large constants, AUI/DAUI/DLUI/DAHI/DATI should be used in a stylized manner.

To create an integer:

```
LUI rd, imm_low
AUI rd, rd, imm_upper
DAHI rd, imm_high
DATI rd, imm_top
```

To create a large offset for a memory access whose address is of the form `rbase+large_offset`:

```
AUI rtmp, rbase, imm_upper
DAHI rtmp, imm_high
DATI rtmp, imm_top
LW rd, (rtmp)imm_low
```

To create a large constant operand for an instruction of the form `rd:=rs+large_immediate` or `rd:=rs-large_immediate`

32-bits:

```
AUI   rtmp, rs,   imm_upper
ADDUI rd,   rtmp, imm_low
```

64-bits:

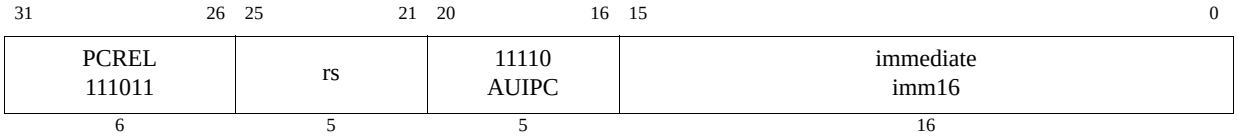
```
AUI  rtmp, rs, imm_upper
DAHI rtmp, imm_high
DATI rtmp, imm_top
DADDUI rd, rtmp,imm_low
```

For other instructions with large constant operands - non-additive instructions such as logical operators AND/OR/XOR, indirect branches, etc. - no idioms are recommended apart from those to create a large constant.

Independent instructions may be interleaved between the instructions in the recommended instruction sequences.

See also PC-relative address computation instructions, such as ADDIUPC, AUIPC, ALUIPC.

DocumentMarginNote



Format: `AUIPC`
`AUIPC rs,imm19` **MIPS32 Release 6**

Purpose: Add Upper Immediate to PC

Description: `GPR[rs] ← memory_address( PC + sign_extend( imm16 << 16 ) )`

The 16 bit immediate is shifted left 16 bits, sign-extended, and added to the address of the AUIPC instruction.

The result of the addition is placed in GPR `rs`.

Restrictions:

Availability:

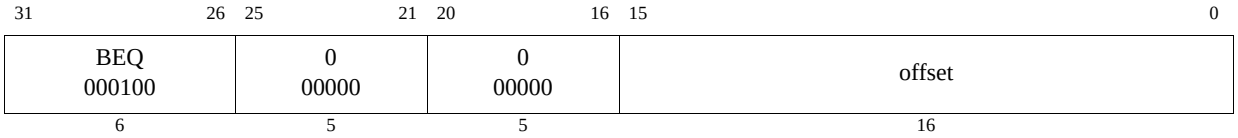
AUIPC is introduced by and required as of MIPS32 Release 6.

Operation

```
GPR[rs] ← memory_address( PC + sign_extend( imm16 << 16 ) )
```

Exceptions:

None.



Format: `B offset` **Assembly Idiom**

Purpose: Unconditional Branch

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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Operation:

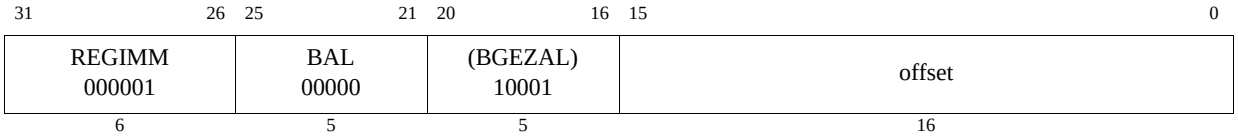
```
I:      target_offset ← sign_extend(offset || 0^2)
I+1:    PC ← PC + target_offset
```

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.



Format: `BAL offset` **MIPS32 Release 6 instruction, pre-MIPS32 Release 6 Assembly Idiom**

Purpose: Branch and Link

To do an unconditional PC-relative procedure call

Description: `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.

Prior to Release 6: BAL offset is the assembly idiom used to denote an unconditional branch. The actual instruction is interpreted by the hardware as BGEZAL r0, offset.

Release 6 keeps the BAL special case of BGEZAL, but removes all other instances of BGEZAL. BGEZAL with `rs` any register other than `GPR[0]` is required to signal a Reserved Instruction exception. Therefore, as of Release 6, BAL is architecturally an instruction supported by the hardware, rather than a special case of BGEZAL.

Restrictions:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

BAL without a corresponding return should NOT be used to read the PC. Doing so is likely to cause a performance loss on processors with a return address predictor.

Operation:

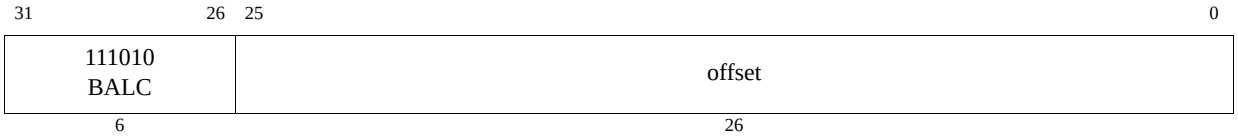
```
I:      target_offset ← sign_extend(offset || 0^2)
        GPR[31] ← PC + 8
I+1:    PC ← PC + target_offset
```

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.



Format: `BALC`
`BALC offset26`

MIPS32 Release 6

Purpose: Branch and Link, Compact

Description: `procedure_call (no delay slot)`

Place the return address link in GPR 31. The return link is the address of the instruction immediately following the branch, where execution continues after a procedure call. (Since compact branches have no delay slots, see below.)

The return address link is updated with the address of the next instruction.

The branch target is formed by sign extending the 26-bit offset field of the instruction shifted left by 2 bits (since MIPS instructions are 4 byte aligned), and adding it to the PC of the following instruction, i.e. adding it to the PC of the current instruction + the instruction length, PC+4.[1]

Compact branches have no delay slot: the instruction after the branch is NOT executed if the branch is taken.

Restrictions:

This instruction is an unconditional, always taken, compact branch, and hence does not have a Forbidden Slot.

Availability:

This instruction is introduced by and required as of MIPS32 Release 6.

Exceptions:

None[2]

Operation:

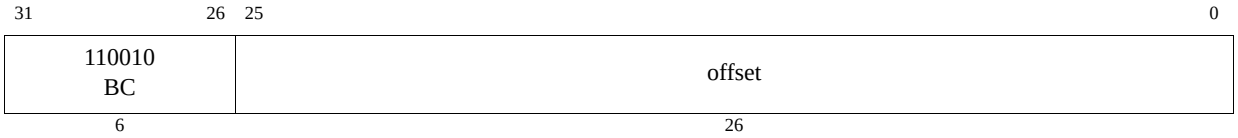
```
target_offset ← sign_extend( offset || 0^2 )
GPR[31] ← PC+4
PC ← memory_address( PC+4 + sign_extend(target_offset) )
```

Special Considerations:

MIPS32 Release 6 instruction `BALC 111010.offset26` occupies the same encoding as pre-Release 6 instruction `SWC2 111010.base.rt.offset16`. The `SWC2 111010.base.rt.offset16` is removed in MIPS32 Release 6.

Note the special restrictions on relative addresses that cross the 2G / 32-bit signed boundaries, formulated as pseudo-code function `memory_address`.

1. In this instruction description, PC refers to the PC of the current instruction. In some older instruction descriptions, PC is used inconsistently, often in association with the I: / I+1: notation related to delayed branches. E.g. in pre-MIPS32 Release 6 branch (B) PC refers to the address of the next instruction, but in branch and link (BAL) it refers to the address of the current instruction.
2. Except for possible Reserved Instruction Exception if used in a Forbidden Slot, true of these and most other control transfer instructions (CTIs)



Format: `BC`
`BC offset26`

MIPS32 Release 6

Purpose: Branch, Compact

Description: `PC ← PC+4 + sign_extend( offset << 2 )`

The branch target is formed by sign extending the 26-bit offset field of the instruction shifted left by 2 bits (since MIPS instructions are 4 byte aligned), and adding it to the PC of the following instruction, i.e. adding it to the PC of the current instruction + the instruction length, PC+4.[1]

Compact branches have no delay slot: the instruction after the branch is NOT executed if the branch is taken.

Restrictions:

This instruction is an unconditional, always taken, compact branch, and hence does not have a Forbidden Slot.

Availability:

This instruction is introduced by and required as of MIPS32 Release 6.

Exceptions:

None[2]

Operation:

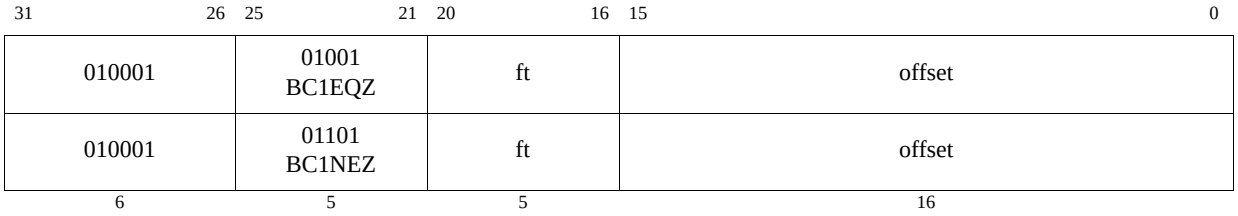
```
target_offset ← sign_extend( offset || 0^2 )
PC ← memory_address( PC+4 + sign_extend(target_offset) )
```

Special Considerations:

MIPS32 Release 6 instruction `BC 110010.offset26` occupies the same encoding as pre-MIPS32 Release 6 instruction `LWC2 110010.base.rt.offset16`. The `LWC2 110010.base.rt.offset16` is removed in MIPS32 Release 6.

Note the special restrictions on relative addresses that cross the 2G / 32-bit signed boundaries, formulated as pseudo-code function `memory_address`.

1. In this instruction description, PC refers to the PC of the current instruction. In some older instruction descriptions, PC is used inconsistently, often in association with the I: / I+1: notation related to delayed branches. E.g. in pre-MIPS32 Release 6 branch (B) PC refers to the address of the next instruction, but in branch and link (BAL) it refers to the address of the current instruction.
2. Except for possible Reserved Instruction Exception if used in a Forbidden Slot, true of these and most other control transfer instructions (CTIs)



31 – 26	25 – 21	20 – 16	15 – 0
010001	01001 BC1EQZ	ft	offset
010001	01101 BC1NEZ	ft	offset
6	5	5	16

Format: `BC1EQZ BC1NEZ`

`BC1EQZ ft, offset16` **MIPS32 Release 6**

`BC1NEZ ft, offset16` **MIPS32 Release 6**

Purpose: Branch if Coprocessor 1 (FPU) Register Bit 0 Equal/Not Equal to Zero

BC1EQZ: Branch if Coprocessor 1 (FPR) Register Bit 0 is Equal to Zero
BC1NEZ: Branch if Coprocessor 1 (FPR) Register Bit 0 is Not Equal to Zero

Description:

```
BC1EQZ:  if FPR[ft].bit0 = 0 then branch
BC1NEZ:  if FPR[ft].bit0 ≠ 0 then branch
```

The condition is evaluated on FPU register *ft*.

For BC1EQZ the condition is true if and only if bit 0 of the FPU register *ft* is zero.

For BC1NEZ the condition is true if and only if bit 0 of the FPU register *ft* is non-zero.

If the condition is false, the branch is not taken, and execution continues with the next instruction.

If the condition is true, the instruction in the branch delay slot is executed, and the branch is taken.

The branch target is formed by sign extending the 16-bit offset field of the instruction shifted left by 2 bits (since MIPS instructions are 4 byte aligned), and adding it to the PC of the following instruction, i.e. adding it to the PC of the current instruction + the instruction length, PC+4.[1] The instruction following the branch is the delay slot of the branch.

Restrictions:

If access to Coprocessor 1 is not enabled, a Coprocessor Unusable Exception is signaled.

Since these instructions BC1EQZ and BC1NEZ do not depend on a particular floating point data type, they operate whenever Coprocessor 1 is enabled. They do NOT produce a Reserved Instruction Exception depending on FIR bits such as FIR.S, FIR.D, FIR.W, FIR.L, etc.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

BC1EQZ and BC1NEZ are introduced by and required as of MIPS32 Release 6.

1. In this instruction description, PC refers to the PC of the current instruction. In some older instruction descriptions, PC is used inconsistently, often in association with the I: / I+1: notation related to delayed branches. E.g. in pre-MIPS32 Release 6 branch (B) PC refers to the address of the next instruction, but in branch and link (BAL) it refers to the address of the current instruction.

Exceptions:

Coprocessor Unusable[2]

Operation:

```
tmp ← ValueFPR(ft, UNINTERPRETED_WORD)
BC1EQZ: cond ← tmp.bit0 = 0
BC1NEZ: cond ← tmp.bit0 ≠ 0
if cond then
   I:     target_PC ← memory_address( PC+4 + sign_extend( offset << 2 )
   I+1:   PC ←target_PC
endif
```

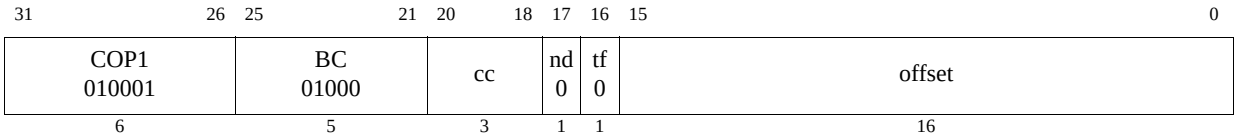
Programming Notes:

As of MIPS32 Release 6 these instructions, BC1EQZ and BC2EQZ, replace the pre-MIPS32 Release 6 instructions BC1F and BC1T, which depended on a 1-bit condition code distinct from the ordinary coprocessor registers. These MIPS32 Release 6 FPU branches depend on bit 0 of the scalar FPU register.

Note: BC1EQZ and BC1NEZ do not have a format or data type width. The same instructions are used for branches based on conditions involving any format, including 32-bit S (single precision) and W (word) format, and 64-bit D (double precision) and L (longword) format, as well as 128-bit MSA. The FPU scalar comparison instruction CMP.condn.fmt produces a mask of the same width as its format. E.g. CMP.condn.S produces a 32-bit mask, zero-extended to 64 bits. Nevertheless, BC1EQZ and BC1NEQZ, considering only bit 0, operate as expected.

I.e. CMP.condn.fmt produces an allZeroes/allOnes truth mask, whereas the BC1* branches consume bit 0 only.

2. In MIPS32 Release 6, BC1EQZ and BC1NEZ are required, if the FPU is implemented. They must not signal a Reserved Instruction Exception. They can signal a Coprocessor Unusable Exception.



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

Purpose: Branch on FP False

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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

This instruction has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the ‘branch on condition 1 equal to zero’ instruction. Refer to the ‘BC1EQZ’ instruction in this manual for more information.

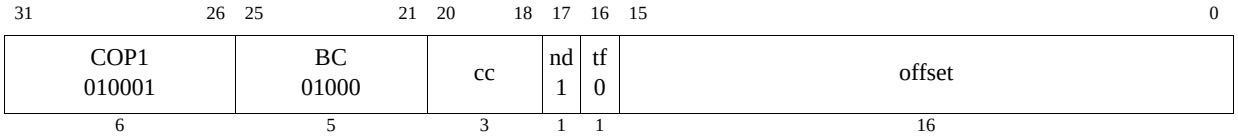
Historical Information:

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.

1. “Removed by MIPS Release 6” means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Branch on FP False Likely

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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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
        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.

The Branch Likely instructions were deprecated prior to Release 1 of the MIPS32 architecture. The Branch Likely instructions were never added to the microMIPS ISA, and have been removed from the base MIPS ISA by Release 6[1].

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.

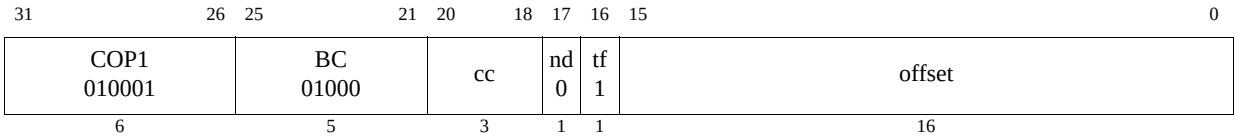
Historical Information:

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 and III architectures, 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.

1. “Removed by MIPS Release 6” means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception, and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Branch on FP True

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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Programming Notes:

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

(JR) instructions to branch to addresses outside this range.

This instruction has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the ‘branch on condition 1 not equal to zero’ instruction. Refer to the ‘BC1NEZ’ instruction in this manual for more information.

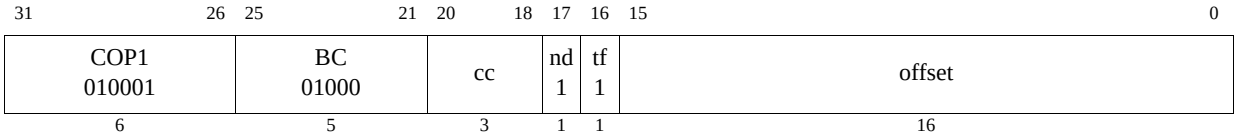
Historical Information:

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.

1. “Removed by MIPS Release 6” means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Branch on FP True Likely

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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
        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.

The Branch Likely instructions were deprecated prior to Release 1 of the MIPS32 architecture. The Branch Likely instructions were never added to the microMIPS ISA, and have been removed from the base MIPS ISA by Release 6[1].

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.

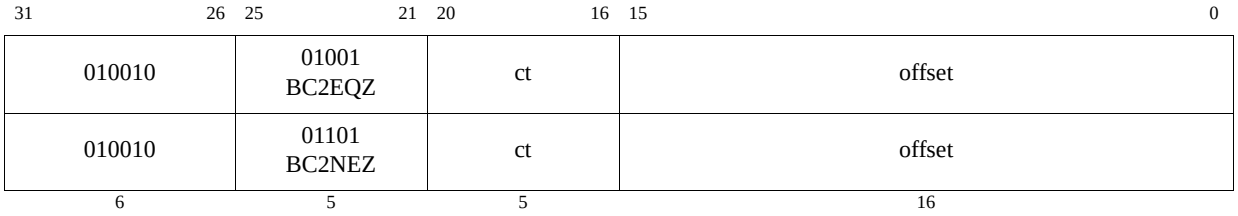
Historical Information:

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 and III architectures, 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.

1. “Removed by MIPS Release 6” means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception, and if the instruction encoding has not been reused for a different instruction.



31–26	25–21	20–16	15–0
010010	01001 BC2EQZ	ct	offset
010010	01101 BC2NEZ	ct	offset
6	5	5	16

Format: `BC2EQZ BC2NEZ`

`BC2EQZ ct, offset16` **MIPS32 Release 6**

`BC2NEZ ct, offset16` **MIPS32 Release 6**

Purpose: Branch if Coprocessor 2 Condition (Register) Equal/Not Equal to Zero

BC2EQZ: Branch if Coprocessor 2 Condition (Register) is Equal to Zero

BC2NEZ: Branch if Coprocessor 2 Condition (Register) is Not Equal to Zero

Description:

```
BC2EQZ:  if COP2Condition[ct] = 0 then branch
BC2NEZ:  if COP2Condition[ct] ≠ 0 then branch
```

The 5-bit field `ct` specifies a coprocessor 2 condition.

For BC2EQZ if the coprocessor 2 condition is true the branch is taken.

For BC2NEZ if the coprocessor 2 condition is false the branch is taken.

BC2EQZ and BC2NEZ have branch delay slots.

The branch target is formed by sign extending the 16-bit offset field of the instruction shifted left by 2 bits (since MIPS instructions are 4 byte aligned), and adding it to the PC of the following instruction, i.e. adding it to the PC of the current instruction + the instruction length, PC+4.[1] The instruction following the branch is the delay slot of the branch.

Restrictions:

If access to Coprocessor 2 is not enabled, a Coprocessor Unusable Exception is signaled.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

BC2EQZ and BC2NEZ are introduced by and required as of MIPS32 Release 6.

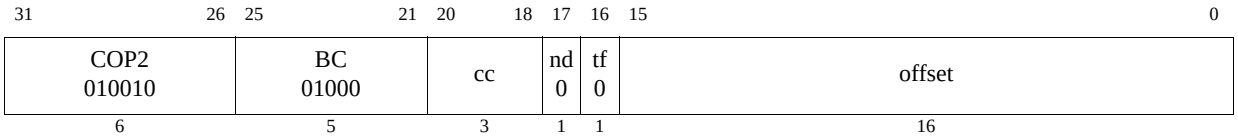
1. In this instruction description, PC refers to the PC of the current instruction. In some older instruction descriptions, PC is used inconsistently, often in association with the I: / I+1: notation related to delayed branches. E.g. in pre-MIPS32 Release 6 branch (B) PC refers to the address of the next instruction, but in branch and link (BAL) it refers to the address of the current instruction.

Exceptions:

Coprocessor Unusable, Reserved Instruction

Operation:

```
tmpcond ← Coprocessor2Condition(ct)
if BC2NEZ then
  tmpcond ← not(tmpcond)
endif

if tmpcond then
  PC ← PC+4 + sign_extend( imm16 << 2) )
endif
```



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

Purpose: Branch on COP2 False

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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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:

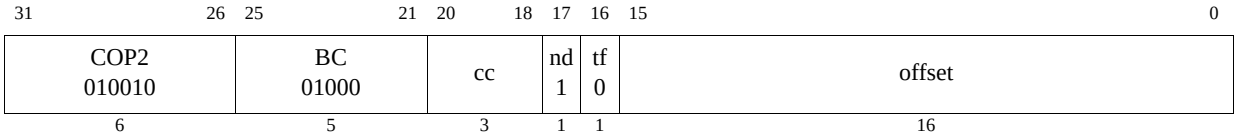
Coprocessor Unusable, Reserved Instruction

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.

This instruction has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the ‘branch on condition 2 equal to zero’ instruction. Refer to the ‘BC2EQZ’ instruction in this manual for more information.

1. “Removed by MIPS Release 6” means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Branch on COP2 False Likely

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

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

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

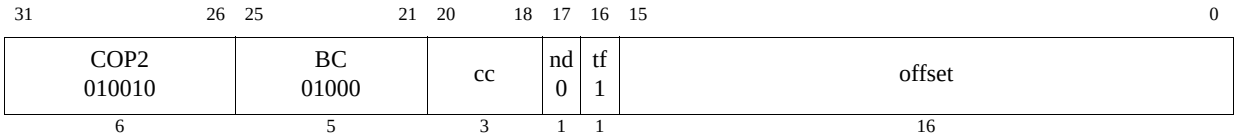
Programming Notes:

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

The Branch Likely instructions were deprecated prior to Release 1 of the MIPS32 architecture. The Branch Likely instructions were never added to the microMIPS ISA, and have been removed from the base MIPS ISA by Release 6[1].

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.

1. "Removed by MIPS Release 6" means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception, and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Branch on COP2 True

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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

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

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

Exceptions:

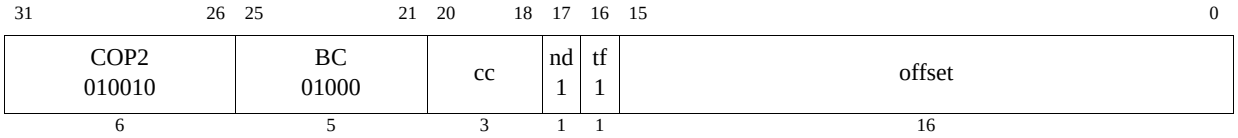
Coprocessor Unusable, Reserved Instruction

Programming Notes:

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

This instruction has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the ‘branch on condition 2 not equal to zero’ instruction. Refer to the ‘BC2NEZ’ instruction in this manual for more information.

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Branch on COP2 True Likely

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

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

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

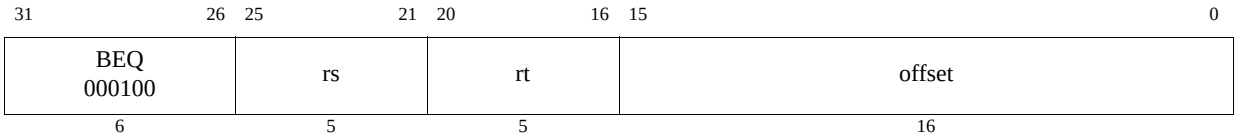
Programming Notes:

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

The Branch Likely instructions were deprecated prior to Release 1 of the MIPS32 architecture. The Branch Likely instructions were never added to the microMIPS ISA, and have been removed from the base MIPS ISA by Release 6[1].

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.

1. "Removed by MIPS Release 6" means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception, and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Branch on Equal

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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Operation:

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

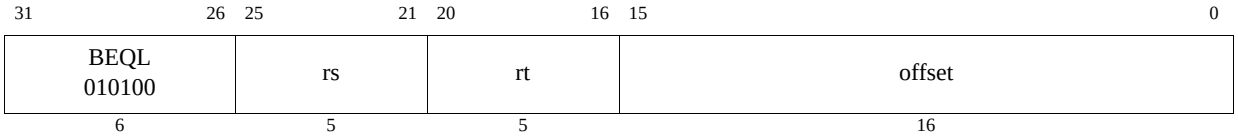
Exceptions:

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.



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

Purpose: Branch on Equal Likely

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

The Branch Likely instructions were deprecated prior to Release 1 of the MIPS32 architecture. The Branch Likely instructions were never added to the microMIPS ISA, and have been removed from the base MIPS ISA by Release 6[1].

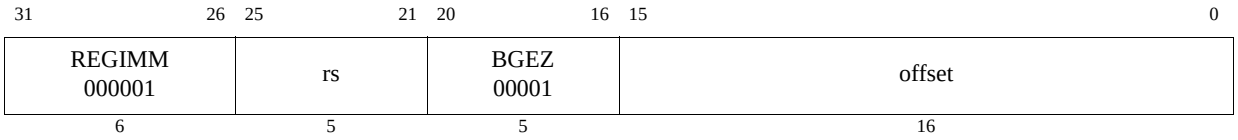
Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability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

---

1. "Removed by MIPS Release 6" means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception, and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Branch on Greater Than or Equal to Zero

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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Operation:

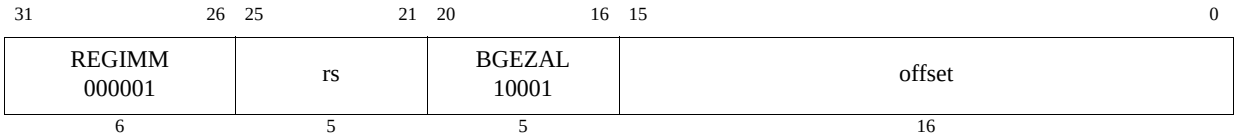
```
I:      target_offset ← sign_extend(offset || 0^2)
        condition ← GPR[rs] ≥ 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_offset
        endif
```

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.



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

Purpose: Branch on Greater Than or Equal to Zero and Link

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:

This instruction has been removed in the MIPS32 Release 6 architecture, with the exception of special case BAL (unconditional Branch and Link) which was an alias for BGEZAL with rs=0.

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.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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
```

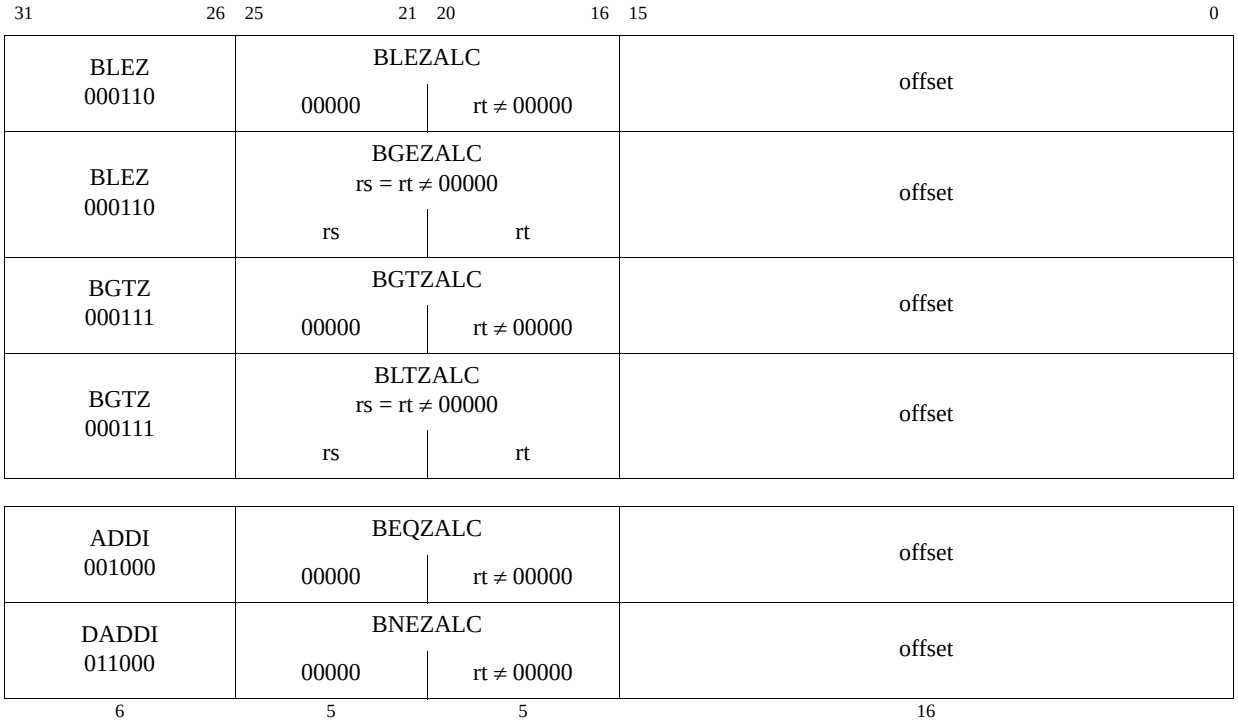
Exceptions:

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.



Format: `B{LE,GE,GT,LT,EQ,NE}ZALC`

```
BLEZALC rt, offset                    MIPS32 Release 6
BGEZALC rt, offset                    MIPS32 Release 6
BGTZALC rt, offset                    MIPS32 Release 6
BLTZALC rt, offset                    MIPS32 Release 6
BEQZALC rt, offset                    MIPS32 Release 6
BNEZALC rt, offset                    MIPS32 Release 6
```

Preliminary

Purpose: Compact zero-compare branch-and-link instructions

BLEZALC: Compact branch-and-link if GPR `rt` is less than or equal to zero

BGEZALC: Compact branch-and-link if GPR `rt` is greater than or equal to zero

BGTZALC: Compact branch-and-link if GPR `rt` is greater than zero

BLTZALC: Compact branch-and-link if GPR `rt` is less than to zero

BEQZALC: Compact branch-and-link if GPR `rt` is equal to zero

BNEZALC: Compact branch-and-link if GPR `rt` is not equal to zero

Description:

The condition is evaluated. If the condition is true, the branch is taken.

Places the return address link in GPR 31. The return link is the address of the instruction immediately following the branch, where execution continues after a procedure call.

The return address link is unconditionally updated.

The branch target is formed by sign extending the 16-bit offset field of the instruction shifted left by 2 bits (since MIPS instructions are 4 byte aligned), and adding it to the PC of the following instruction, i.e. adding it to the PC of the current instruction + the instruction length, PC+4.[1]

Compact branches have no delay slot: the instruction after the branch is NOT executed if the branch is taken, but it and following instructions are executed if the branch is not-taken.

The instruction after the branch is defined to be the “Forbidden Slot”. If the compact branch is not taken, the instruction in the forbidden slot is executed. See below for more details.

BLEZALC: the condition is true if and only if GPR `rt` is less than or equal to zero.
BGEZALC: the condition is true if and only if GPR `rt` is greater than or equal to zero.
BLTZALC: the condition is true if and only if GPR `rt` is less than zero.
BGTZALC: the condition is true if and only if GPR `rt` is greater than zero.
BEQZALC: the condition is true if and only if GPR `rt` is equal to zero.
BNEZALC: the condition is true if and only if GPR `rt` is not equal to zero.

Restrictions:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Release 6: If a control transfer instruction (CTI) is placed in the forbidden slot of a branch or jump, Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

These instructions are introduced by and required as of MIPS32 Release 6.

Exceptions:

None[2]

Operation:

```
GPR[31] ← PC+4
target_offset ← sign_extend( offset || 0^2 )

BLTZALC: cond ← GPR[rt] < 0
BLEZALC: cond ← GPR[rt] ≤ 0
BGEZALC: cond ← GPR[rt] ≥ 0
BGTZALC: cond ← GPR[rt] > 0
BEQZALC: cond ← GPR[rt] = 0
BNEZALC: cond ← GPR[rt] ≠ 0

if cond then
  PC ← memory_address( PC+4+ sign_extend( target_offset ) )
endif
```

Special Considerations:

1. In this instruction description, PC refers to the PC of the current instruction. In some older instruction descriptions, PC is used inconsistently, often in association with the I: / I+1: notation related to delayed branches. E.g. in pre-MIPS32 Release 6 branch (B) PC refers to the address of the next instruction, but in branch and link (BAL) it refers to the address of the current instruction.
2. Except for possible Reserved Instruction Exception if used in a Forbidden Slot, true of these and most other control transfer instructions (CTIs)

See section A.4 on page 763 in Volume II for a complete overview of MIPS32 Release 6 instruction encodings. Brief notes related to these instructions:

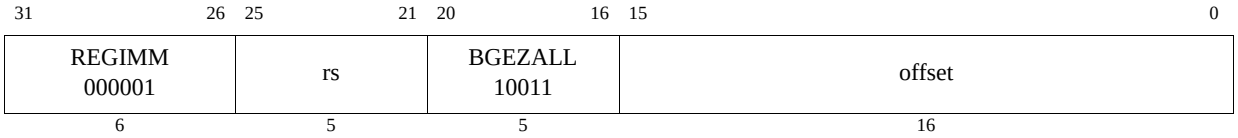
These instructions occupy primary opcode spaces originally allocated to other instructions. BLEZALC and BGEZALC have the same primary opcode as BLEZ, and are distinguished by rs and rt register numbers. Similarly, BGTZALC and BLTZALC have the same primary opcode as BGTZ, and are distinguished by register fields. BEQZALC and BNEZALC reuse the primary opcodes ADDI and DADDI.

In order to provide a comprehensive set of compare-and-branch instructions, MIPS32 Release 6 reduces some of the redundancy in instruction encoding related to register numbers. This is indicated by constraints applied to the register encodings. For example, `BGEZC rs,rt` is encoded as `010110.rs.rt.offset16`, with constraints `rs≠00000`, `rs≠00000` and `rs=rt` (since `BGEZ` with both operands the same register is always true, hence redundant). Similarly, `BEQC rs,rt` has the constraints `rs≠00000`, `rt≠00000`, and `rs<rt` (since `BEQC` with both operands the same register is always true, hence redundant). Note that "`rs`" and "`rt`" in these constraints refer to register numbers as encoded within the instruction, not the values contained by the registers.

Programming Notes:

Old software that performs incomplete instruction decode may incorrectly decode these new instructions, because of their very tight encoding. E.g. a disassembler that looks only at the primary opcode field, instruction bits 31-26, to decode BLEZL 010110.rs.00000.offset16, without checking that the "rt" field is zero. Such software violated the pre-MIPS32 Release 6 architecture specification.

With the 16-bit offset shifted left 2 bits and sign extended, the conditional branch range is ± 128 KBytes. Other instructions such as pre-MIPS32 Release 6 JAL and JALR, or MIPS32 Release 6 JIALC and BALC have larger ranges. In particular, BALC, with a 26-bit offset shifted by 2 bits, has a 28-bit range, ± 128 MBytes. Code sequences using AUIPC, DAHI, DATI, and JIALC allow still greater PC-relative range.



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

Purpose: Branch on Greater Than or Equal to Zero and Link Likely

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

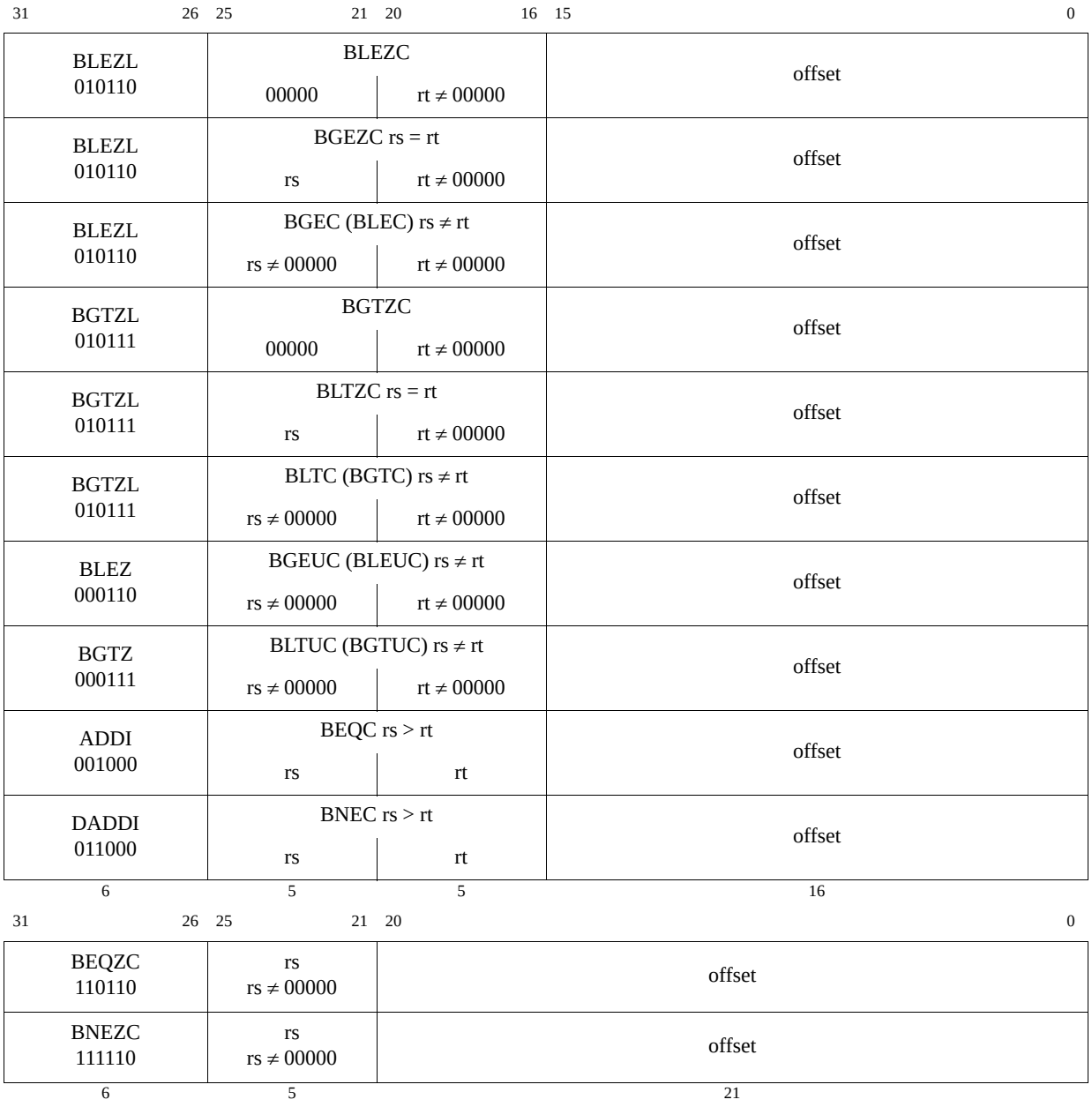
The Branch Likely instructions were deprecated prior to Release 1 of the MIPS32 architecture. The Branch Likely instructions were never added to the microMIPS ISA, and have been removed from the base MIPS ISA by Release 6[1].

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.

1. "Removed by MIPS Release 6" means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception, and if the instruction encoding has not been reused for a different instruction.



31–26	25–21	20–16	15–0
BLEZL 010110	BLEZC 00000	rt ≠ 00000	offset
BLEZL 010110	BGEZC rs = rt rs	rt ≠ 00000	offset
BLEZL 010110	BGEC (BLEC) rs ≠ rt rs ≠ 00000	rt ≠ 00000	offset
BGTZL 010111	BGTZC 00000	rt ≠ 00000	offset
BGTZL 010111	BLTZC rs = rt rs	rt ≠ 00000	offset
BGTZL 010111	BLTC (BGTC) rs ≠ rt rs ≠ 00000	rt ≠ 00000	offset
BLEZ 000110	BGEUC (BLEUC) rs ≠ rt rs ≠ 00000	rt ≠ 00000	offset
BGTZ 000111	BLTUC (BGTUC) rs ≠ rt rs ≠ 00000	rt ≠ 00000	offset
ADDI 001000	BEQC rs > rt rs	rt	offset
DADDI 011000	BNEC rs > rt rs	rt	offset
6	5	5	16

31–26	25–21	20–0
BEQZC 110110	rs rs ≠ 00000	offset
BNEZC 111110	rs rs ≠ 00000	offset
6	5	21

B{LEZ,GEZ,GTZ,GE,LT,LTZ,GEU,LTU,EQ,NE,EQZ,NEZ}C

Purpose: Compact compare-branch instructions

Description: `if condition GPR(s) then compact branch (no delay slot)`

Format:

Signed register-register compare and branch with 16-bit offset:

```
BLTC rs,rt, offset                    MIPS32 Release 6
BLEC rs,rt, offset                    MIPS32 Release 6
BGEC rs,rt, offset                    MIPS32 Release 6
BGTC rs,rt, offset                    MIPS32 Release 6
BLTUC rs,rt, offset                   MIPS32 Release 6
BLEUC rs,rt, offset                   MIPS32 Release 6
BGEUC rs,rt, offset                   MIPS32 Release 6
BGTUC rs,rt, offset                   MIPS32 Release 6
BEQC rs,rt, offset                    MIPS32 Release 6
BNEC rs,rt, offset                    MIPS32 Release 6
```

Unsigned register-register compare and branch with 16-bit offset:

```
BLTUC rs,rt, offset                   MIPS32 Release 6
BLEUC rs,rt, offset                   MIPS32 Release 6
BGEUC rs,rt, offset                   MIPS32 Release 6
BGTUC rs,rt, offset                   MIPS32 Release 6
```

Compare register to zero and branch with 16-bit offset:

```
BLTZC rt, offset                      MIPS32 Release 6
BLEZC rt, offset                      MIPS32 Release 6
BGEZC rt, offset                      MIPS32 Release 6
BGTZC rt, offset                      MIPS32 Release 6
```

Compare register to zero and branch with 21-bit offset:

```
BEQZC rs, offset                      MIPS32 Release 6
BNEZC rs, offset                      MIPS32 Release 6
```


Purpose: Compact compare-branch instructions

Signed register-register compare and branch with 16-bit offset:

- BLTC: Compact branch if GPR `rs` is less than GPR `rt`
- BLEC: Compact branch if GPR `rt` is greater than or equal to GPR `rs` (alias for BGEC)
- BGEC: Compact branch if GPR `rs` is greater than or equal to GPR `rt`
- BGTC: Compact branch if GPR `rt` is less than GPR `rs` (alias for BLTC)
- BEQC: Compact branch if GPRs are equal
- BNEC: Compact branch if GPRs are not equal

Unsigned register-register compare and branch with 16-bit offset:

- BLTUC: Compact branch if GPR `rs` is less than or equal to GPR `rt`, unsigned
- BLEUC: Compact branch if GPR `rt` is less than GPR `rt`, unsigned (alias for BLEUC)
- BGEUC: Compact branch if GPR `rs` is less than GPR `rt`, unsigned
- BGTUC: Compact branch if GPR `rt` is less than or equal to GPR `rs`, unsigned (alias for BGTUC)

Compare register to zero and branch with 21-bit offset:

- BLTZC: Compact branch if GPR is less than zero
- BLEZC: Compact branch if GPR less than or equal to zero
- BGEZC: Compact branch if GPR greater than or equal to zero
- BGTZC: Compact branch if GPR greater than zero

Compare register to zero and branch with 21-bit offset:

BEQZC: Compact branch if GPR is equal to zero (21-bit offset)

BNEZC: Compact branch if GPR is not equal to zero (21-bit offset)

Description: `if condition GPR(s) then compact branch (no delay slot):`

The condition is evaluated. If the condition is true, the branch is taken.

The branch target is formed by sign extending the bit offset field of the instruction shifted left by 2 bits (since MIPS instructions are 4 byte aligned), and adding it to the PC of the following instruction, i.e. adding it to the PC of the current instruction + the instruction length, PC+4.[1]

The offset is 16 bits for most compact branches, including BLTC, BLEC, BGEC, BGTC, NEQC, BNEC, BLTUC, BLEUC, BGEUC, BGTC, BLTZC, BLEZC, BGEZC, BGTZC. The offset is 21 bits for BEQZC and BNEZC.

Compact branches have no delay slot: the instruction after the branch is NOT executed if the branch is taken.

The instruction after the branch is defined to be the “Forbidden Slot”. If the compact branch is not taken, the instruction in the forbidden slot is executed. See below for more details.

BLTC: `if GPR[rs] < GPR[rt] then branch`
BGEC: `if GPR[rs] ≥ GPR[rt] then branch`
BEQC: `if GPR[rs] = GPR[rt] then branch`
BNEC: `if GPR[rs] ≠ GPR[rt] then branch`

BLTUC: `if GPR[rs] = GPR[rt] then branch`, unsigned
BGEUC: `if GPR[rs] = GPR[rt] then branch`, unsigned

BLTZC: `if GPR[rt] < 0 then branch`
BLEZC: `if GPR[rt] ≤ 0 then branch`
BGEZC: `if GPR[rt] ≥ 0 then branch`
BGTZC: `if GPR[rt] = 0 then branch`

BEQZC: `if GPR[rs] = 0 then branch` (21-bit offset)
BNEZC: `if GPR[rs] ≠ 0 then branch` (21-bit offset)

Reversing operands `rs` and `rt` allows other comparisons to be synthesized:
BLEC: `if GPR[rt] = GPR[rs] then branch` (alias for BGEC)
BGTC: `if GPR[rt] = GPR[rs] then branch`(alias for BLTC)
BLEUC: `if GPR[rt] = GPR[rs] then branch`, unsigned (alias for BGEUC)
BGTUC: `if GPR[rt] = GPR[rs] then branch`, unsigned (alias for BLTUC)

Restrictions:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Release 6: If a control transfer instruction (CTI) is placed in the forbidden slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

1. In this instruction description, PC refers to the PC of the current instruction. In some older instruction descriptions, PC is used inconsistently, often in association with the I: / I+1: notation related to delayed branches. E.g. in pre-MIPS32 Release 6 branch (B) PC refers to the address of the next instruction, but in branch and link (BAL) it refers to the address of the current instruction.

Availability:

These instructions are introduced by and required as of MIPS32 Release 6.

Exceptions:

None[2]

Operation:

```
target_offset ← sign_extend( offset || 0^2 )

/* Register-register compare and branch, 16 bit offset: */
BLTC: cond ← GPR[rs] < GPR[rt]
BGEC: cond ← GPR[rs] ≥ GPR[rt]
BEQC: cond ← GPR[rs] = GPR[rt]
BNEC: cond ← GPR[rs] ≠ GPR[rt]

/* Unsigned: */
BLTUC: cond ← unsigned(GPR[rs]) < unsigned(GPR[rt])
BGEUC: cond ← unsigned(GPR[rs]) ≥ unsigned(GPR[rt])
/* Compare register to zero: */
BLTZC: cond ← GPR[rt] < 0
BLEZC: cond ← GPR[rt] ≤ 0
BGEZC: cond ← GPR[rt] ≥ 0
BGTZC: cond ← GPR[rt] > 0
/* Compare register to zero, large offset: */
BEQZC: cond ← GPR[rs] = 0
BNEZC: cond ← GPR[rs] ≠ 0

if cond then
  PC ← memory_address( PC+4+ sign_extend( target_offset ) )
endif
```

Special Considerations:

See section A.4 on page 763 in Volume II for a complete overview of MIPS32 Release 6 instruction encodings. Brief notes related to these instructions:

These instructions occupy primary opcode spaces originally allocated to other instructions. In most cases, register fields distinguish these instructions from other instructions using the same primary opcode. The original ADDI and DADDI primary opcodes are reused: MIPS32 Release 6 removes these instructions.

In order to provide a full set of compare-and-branch instructions, MIPS32 Release 6 reduces some of the redundancy in instruction encoding related to register numbers. This is indicated by constraints applied to the register encodings. For example, `BGEZC rs,rt` is encoded as `010110.rs.rt.offset16`, with constraints `rs≠00000`, `rt≠00000` and `rs=rt` (since `BGEZ` with both operands the same register is always true, hence redundant). Similarly, `BEQC rs,rt` has the constraints `rs≠00000`, `rt≠00000`, and `rs<rt` (since `BEQC` with both operands the same register is always true, hence redundant). Note that "`rs`" and "`rt`" in these constraints refer to register numbers as encoded within the instruction, not the values contained by the registers.

Table 3.1 below shows that the compact branches provide a full set of integer comparisons. It is necessary to reverse the operands of the signed and unsigned register-register compares in the shaded cases. Separate instructions are required for comparisons against 0, because the GPR[0] encoding is not allowed in the register-register comparisons,

2. Except for possible Reserved Instruction Exception if used in a Forbidden Slot, true of these and most other control transfer instructions (CTIs)

to save encoding space. Assembler idioms may provide the “reversed” instructions.

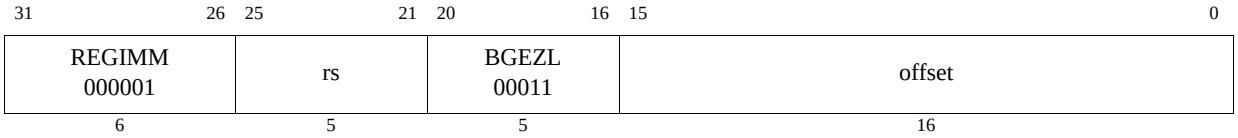
Table 3.1 Compact branches provide a full set of comparisons

register-register comparisons			compare to zero	
operation	**signed**	**unsigned**	**instruction**	**operation**
a < b	BLTC	BLTUC	BLTZC	a < 0
a ≤ b	BLEC = BGEC reversed	BLEUC = BGEUC reversed	BLEZC	a ≤ 0
a = b	BEQC		BEQZC	a = 0
a ≠ b	BNEC		BNEZC	a ≠ 0
a ≥ b	BGEC	BGEUC	BGEZC	a ≥ 0
a > b	BGTC = BLTC reversed	BGTUC = BLTUC reversed	BGTZC	a > 0

Programming Notes:

Old software that performs incomplete instruction decode may incorrectly decode these new instructions, because of their very tight encoding. E.g. a disassembler that looks only at the primary opcode field, instruction bits 31-26, to decode BLEZL 010110.rs.00000.offset16, without checking that the “rt” field is zero. Such software violated the pre-MIPS32 Release 6 architecture specification.

B{LEZ,GEZ,GTZ,GE,LT,LTZ,GEU,LTU,EQ,NE,EQZ,NEZ}C Compare register to zero and branch with 21-bit



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

Purpose: Branch on Greater Than or Equal to Zero Likely

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

The Branch Likely instructions were deprecated prior to Release 1 of the MIPS32 architecture. The Branch Likely instructions were never added to the microMIPS ISA, and have been removed from the base MIPS ISA by Release 6[1].

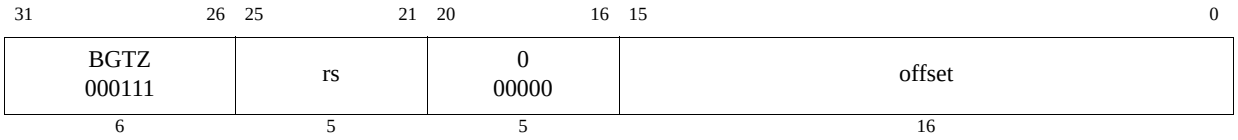
Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability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

---

1. "Removed by MIPS Release 6" means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception, and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Branch on Greater Than Zero

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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Operation:

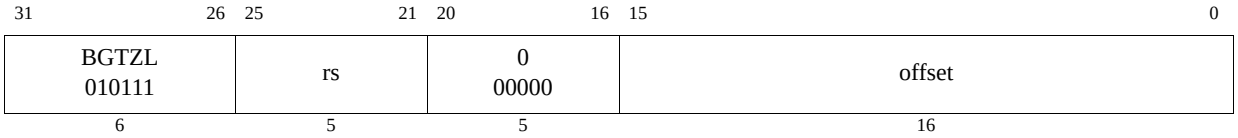
```
I:      target_offset ← sign_extend(offset || 0^2)
        condition ← GPR[rs] > 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_offset
        endif
```

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.



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

Purpose: Branch on Greater Than Zero Likely

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

The Branch Likely instructions were deprecated prior to Release 1 of the MIPS32 architecture. The Branch Likely instructions were never added to the microMIPS ISA, and have been removed from the base MIPS ISA by Release 6[1].

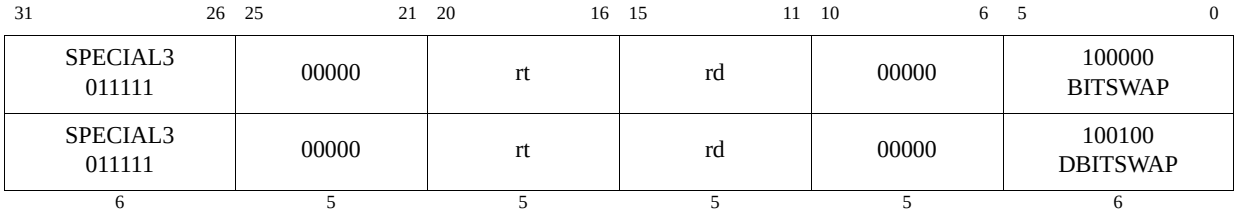
Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability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

---

1. "Removed by MIPS Release 6" means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception, and if the instruction encoding has not been reused for a different instruction.



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL3 011111	00000	rt	rd	00000	100000 BITSWAP
SPECIAL3 011111	00000	rt	rd	00000	100100 DBITSWAP
6	5	5	5	5	6

Format: `BITSWAP DBITSWAP`
`BITSWAP rd,rt` **MIPS32 Release 6**
`DBITSWAP rd,rt` **MIPS64 Release 6**

Purpose: Swaps (reverses) bits in each byte

Description: `GPR[rd].byte(i) ← reverse_bits_in_byte(GPR[rt].byte(i)), for all i`

Each byte in input GPR `rt` is moved to the same byte position in output GPR `rd`, with bits in each byte reversed.

BITSWAP is a 32-bit instruction. BITSWAP operates on all 4 bytes of a 32-bit GPR on a 32-bit CPU. On a 64-bit CPU, BITSWAP operates on the low 4 bytes, sign extending to 64-bits.

DBITSWAP operates on all 8 bytes of a 64-bit GPR on a 64-bit CPU.

Restrictions:

Availability:

The BITSWAP instruction is introduced by and required as of MIPS32 Release 6.

The DBITSWAP instruction is introduced by and required as of MIPS64 Release 6.

Programming Notes:

In MIPS32 Release 6 BITSWAP corresponds to the pre-MIPS32 Release 6 DSP Module BITREV instruction, except that the latter bit-reverses the least-significant 16-bit halfword of the input register, zero extending the rest, while BITSWAP operates on 32-bits. Similarly DBITSWAP operates on 64-bits.

Operation

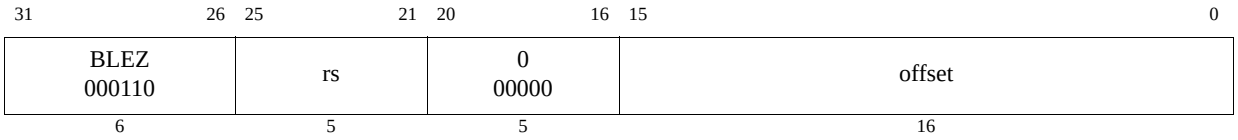
```
BITSWAP:
   for i in 0 to 3 do   /* for all bytes in 32-bit GPR width */
      tmp.byte(i) ← reverse_bits_in_byte( GPR[rt].byte(i) )
   endfor
   GPR[rd] ← sign_extend.32( tmp )

DBITSWAP:
   for i in 0 to 7 do   /* for all bytes in 64-bit GPR width */
      tmp.byte(i) ← reverse_bits_in_byte( GPR[rt].byte(i) )
   endfor
   GPR[rd] ← tmp
```

```
where

   function reverse_bits_in_byte(inbyte)
      outbyte7 ← inbyte0
      outbyte6 ← inbyte1
      outbyte5 ← inbyte2
      outbyte4 ← inbyte3
      outbyte3 ← inbyte4
      outbyte2 ← inbyte5
      outbyte1 ← inbyte6
      outbyte0 ← inbyte7
   end function
```

Exceptions:

BITSWAP: None

DBITSWAP: Reserved Instruction



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

Purpose: Branch on Less Than or Equal to Zero

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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Operation:

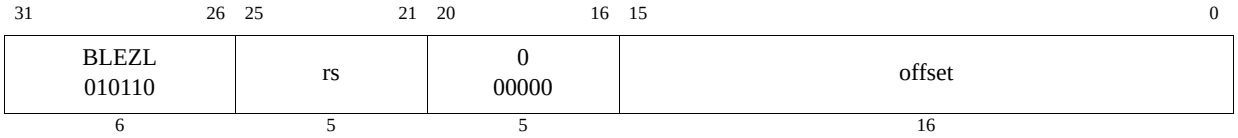
```
I:      target_offset ← sign_extend(offset || 0^2)
        condition ← GPR[rs] ≤ 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_offset
        endif
```

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.



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

Purpose: Branch on Less Than or Equal to Zero Likely

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

The Branch Likely instructions were deprecated prior to Release 1 of the MIPS32 architecture. The Branch Likely instructions were never added to the microMIPS ISA, and have been removed from the base MIPS ISA by Release

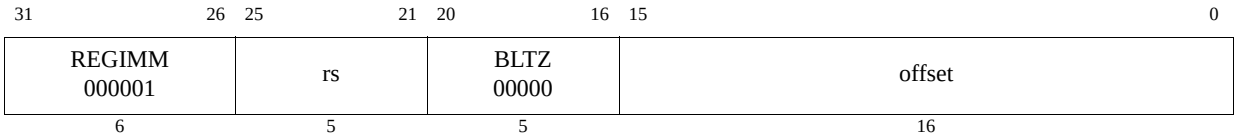
6[1].

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.

1. "Removed by MIPS Release 6" means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception, and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Branch on Less Than Zero

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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Operation:

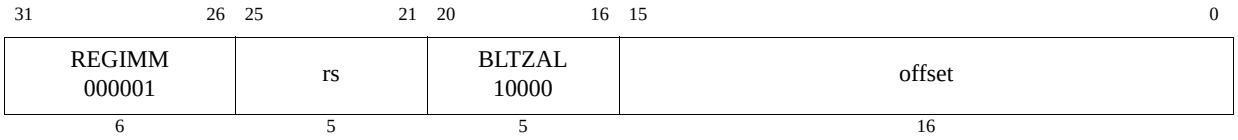
```
I:      target_offset ← sign_extend(offset || 0^2)
        condition ← GPR[rs] < 0^GPRLEN
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

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.



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

Purpose: Branch on Less Than Zero and Link

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:

This instruction has been removed in the MIPS32 Release 6 architecture, with the exception of the special case NAL (Nop and Link) = BLTZAL with rs=0.

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.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

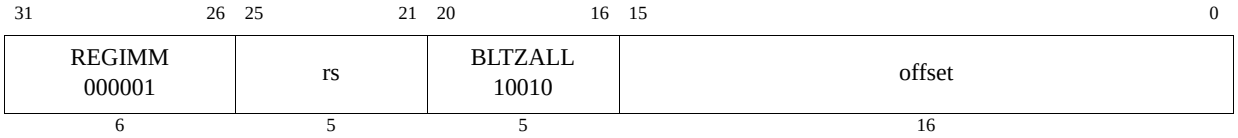
```
I:      target_offset ← sign_extend(offset || 0^2)
        condition ← GPR[rs] < 0^GPRLEN
        GPR[31] ← PC + 8
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

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: Branch on Less Than Zero and Link Likely

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

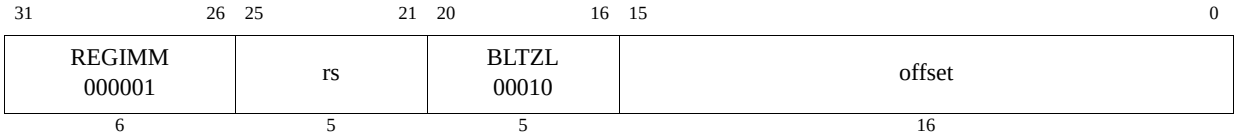
The Branch Likely instructions were deprecated prior to Release 1 of the MIPS32 architecture. The Branch Likely instructions were never added to the microMIPS ISA, and have been removed from the base MIPS ISA by Release 6[1].

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.

1. "Removed by MIPS Release 6" means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception, and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Branch on Less Than Zero Likely

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

The Branch Likely instructions were deprecated prior to Release 1 of the MIPS32 architecture. The Branch Likely instructions were never added to the microMIPS ISA, and have been removed from the base MIPS ISA by Release 6[1].

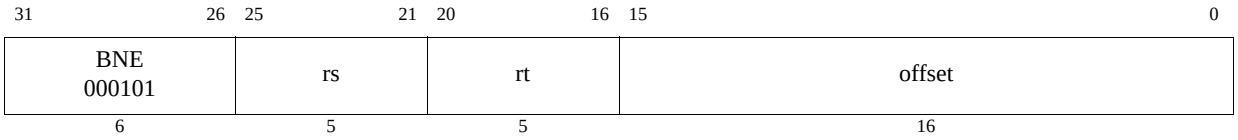
Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability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

---

1. "Removed by MIPS Release 6" means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception, and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Branch on Not Equal

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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Operation:

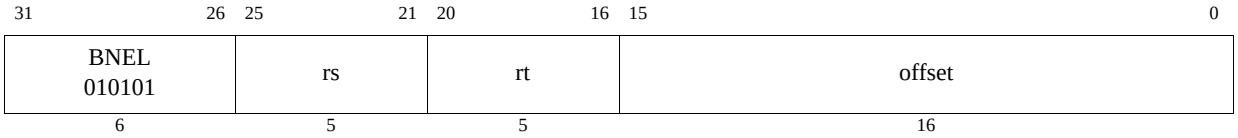
```
I:      target_offset ← sign_extend(offset || 0^2)
        condition ← (GPR[rs] ≠ GPR[rt])
I+1:    if condition then
            PC ← PC + target_offset
        endif
```

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.



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

Purpose: Branch on Not Equal Likely

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

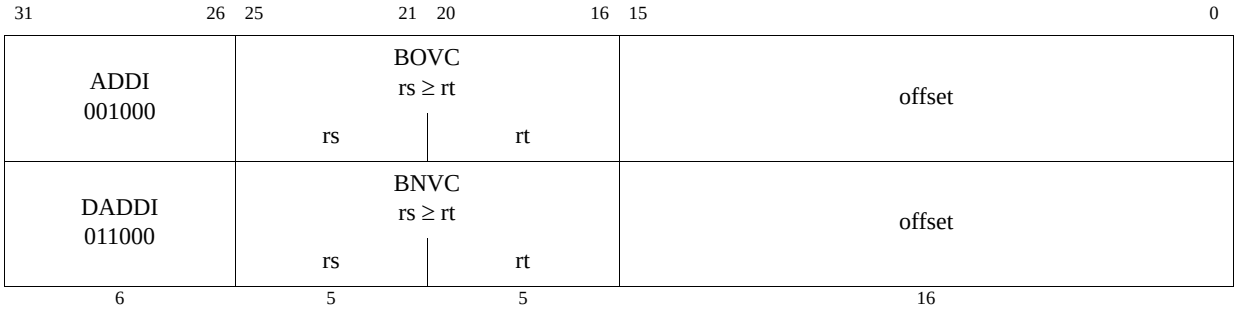
The Branch Likely instructions were deprecated prior to Release 1 of the MIPS32 architecture. The Branch Likely instructions were never added to the microMIPS ISA, and have been removed from the base MIPS ISA by Release 6[1].

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.

1. "Removed by MIPS Release 6" means that implementations of MIPS Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception, and if the instruction encoding has not been reused for a different instruction.



Format: `BOVC BNVC`

`BOVC rs,rt,offset16` **MIPS32 Release 6**

`BNVC rs,rt,offset16` **MIPS32 Release 6**

Purpose: Branch on Overflow, Compact; Branch on No Overflow, Compact

BOVC: Detect overflow for add (signed 32 bits) and branch if overflow.

BNVC: Detect overflow for add (signed 32 bits) and branch if no overflow.

Description:

BOVC performs a signed 32-bit addition of `rs` and `rt`. BOVC discards the sum, but detects signed 32-bit integer overflow, and branches if such overflow is detected.

BNVC performs a signed 32-bit addition of `rs` and `rt`. BNVC discards the sum, but detects signed 32-bit integer overflow, and branches if such overflow is not detected.

BOVC and BNVC are compact branches - they have no branch delay slots, but do have a forbidden slot.

The branch target is formed by sign extending the 16-bit offset field of the instruction shifted left by 2 bits (since MIPS instructions are 4 byte aligned), and adding it to the PC of the following instruction, i.e. adding it to the PC of the current instruction + the instruction length, PC+4.[1]

On 64-bit processors BOVC and BNVC detect signed 32-bit overflow on the input registers as well as the output. This checking is performed even if 64-bit operation is not enabled.

The special case with `rs`=0 is allowed. It is a NOP on MIPS32, but checks that `rt` is a signed 32-bit value on MIPS64.

Restrictions:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

MIPS Release 6: If a control transfer instruction (CTI) is placed in the forbidden slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

BOVC and BNVC are introduced by and required as of MIPS32 Release 6.

1. In this instruction description, PC refers to the PC of the current instruction. In some older instruction descriptions, PC is used inconsistently, often in association with the I: / I+1: notation related to delayed branches. E.g. in pre-MIPS32 Release 6 branch (B) PC refers to the address of the next instruction, but in branch and link (BAL) it refers to the address of the current instruction.

MIPS32 Release 6 BOVC and BNVC replace the pre-MIPS32 Release 6 ADDI (add immediate with overflow trapping) instruction.

Operation

```
input_overflow ← NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])

temp1 ← sign_extend.32( GPR[rs]31..0 )
temp2 ← sign_extend.32( GPR[rs]31..0 )
tempd ← temp1 + temp2 // wider than 32-bit precision
sum_overflow ← (tempd32 ≠ tempd31)

BOVC: cond ← sum_overflow or input_overflow
BNVC: cond ← not( sum_overflow or input_overflow )

if cond then
   PC ← adress_extend( PC+4 + sign_extend.18( offset << 2 ) )
endif
```

Exceptions:

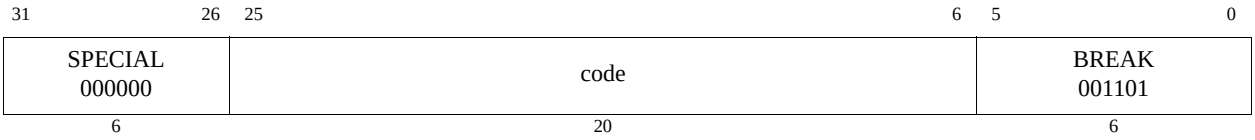
None

Special Considerations:

See section A.4 on page 763 in Volume II for a complete overview of MIPS32 Release 6 instruction encodings. Brief notes related to these instructions:

BOVC uses the primary opcode allocated to ADDI pre-MIPS32 Release 6. MIPS32 Release 6 reuses the ADDI primary opcode for BOVC and other instructions, distinguished by register numbers.

BNVC uses the primary opcode allocated to DADDI pre-MIPS32 Release 6. MIPS32 Release 6 reuses the DADDI primary opcode for BNVC and other instructions, distinguished by register numbers.



Format: `BREAK` **MIPS32**

Purpose: Breakpoint

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:

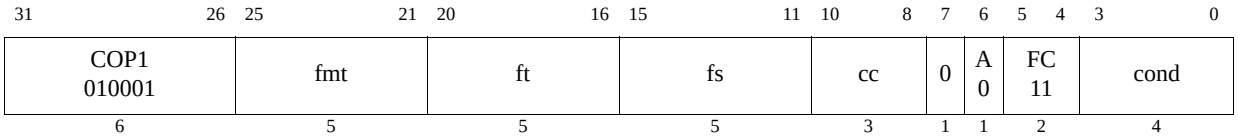
None

Operation:

```
SignalException(Breakpoint)
```

Exceptions:

Breakpoint



Format:

```
C.cond.fmt
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: Floating Point Compare

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

Description: `FPConditionCode(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 the *cond* field of the instruction 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.

In the *cond* field of the instruction: $cond_{2..1}$ specify the nature of the comparison (equals, less than, and so on); $cond_0$ specifies whether the comparison is ordered or unordered, i.e. false or true if any operand is a NaN; $cond_3$ indicates whether the instruction should signal an exception on QNaN inputs, or not (see Table 3.26).

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 Table 3.25. 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 3.26 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 3.2 FPU Comparisons Without Special Operand Exceptions

Instruction	Comparison Predicate					Comparison CC Result		Instruction	
Cond Mnemonic	Name of Predicate and Logically Negated Predicate (Abbreviation)	Relation Values				If Predicate Is True	Inv Op Excp. if QNaN?	Condition Field	
		>	<	=	?			3	2..0
F	False [this predicate is always False]	F	F	F	F	F	No	0	0
	True (T)	T	T	T	T				
UN	Unordered	F	F	F	T	T			1
	Ordered (OR)	T	T	T	F	F			
EQ	Equal	F	F	T	F	T			2
	Not Equal (NEQ)	T	T	F	T	F			
UEQ	Unordered or Equal	F	F	T	T	T			3
	Ordered or Greater Than or Less Than (OGL)	T	T	F	F	F			
OLT	Ordered or Less Than	F	T	F	F	T			4
	Unordered or Greater Than or Equal (UGE)	T	F	T	T	F			
ULT	Unordered or Less Than	F	T	F	T	T			5
	Ordered or Greater Than or Equal (OGE)	T	F	T	F	F			
OLE	Ordered or Less Than or Equal	F	T	T	F	T			6
	Unordered or Greater Than (UGT)	T	F	F	T	F			
ULE	Unordered or Less Than or Equal	F	T	T	T	T			7
	Ordered or Greater Than (OGT)	T	F	F	F	F			
Key: ? = *unordered*, > = *greater than*, < = *less than*, = is *equal*, T = *True*, F = False									

Table 3.3 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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU,.

The result of C.cond.PS is **UNPREDICTABLE** if the condition code number is odd.

This instruction has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the 'CMP.cond.fmt' instruction. Refer to the CMP.cond.fmt instruction in this manual for more information. Note that MIPS32 Release 6 does not support Paired Single (PS).

Availability:

This instruction has been removed in the Release 6 architecture.

1. "Removed by MIPS32 Release 6" means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.

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

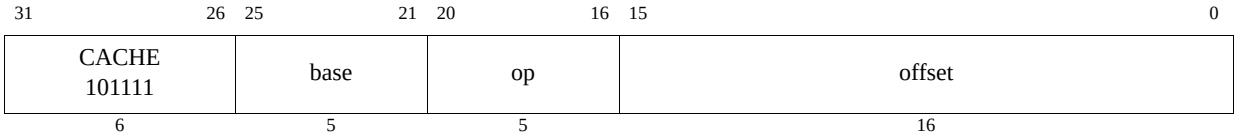
Programming Notes:

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.

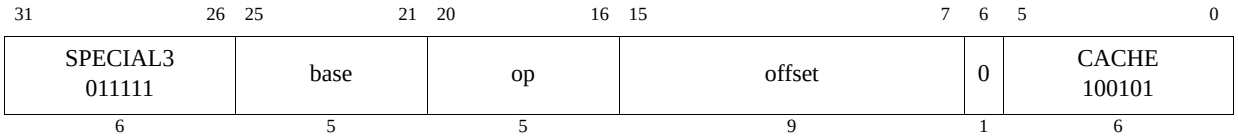
```
# comparisons using explicit tests for QNaN
   c.eq.d $f2,$f4   # check for equal
   nop
   bc1t   L2        # it is equal
   c.un.d $f2,$f4   # it is not equal,
                    # but might be unordered
   bc1t   ERROR     # unordered goes off to an error handler
# not-equal-case code here
   ...
# 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
```

MIPS32, (all release levels less than Release 6)



MIPS32 Release 6 (Release 6 and future)



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

Purpose: Perform Cache Operation

To perform the cache operation specified by op.

Description:

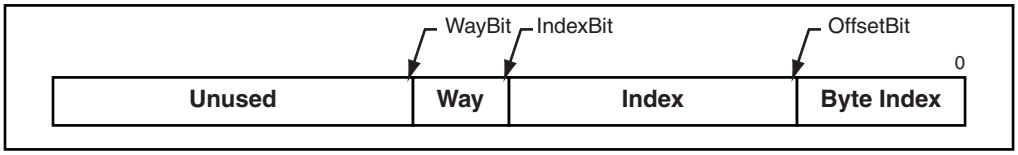
The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

Table 3.4 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, an unmapped address (such as within kseg0) 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 3.3 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. This instruction never causes Execute-Inhibit nor Read-Inhibit exceptions.

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.

The CACHE instruction and the memory transactions which are sourced by the CACHE instruction, such as cache refill or cache writeback, obey the ordering and completion rules of the SYNC instruction.

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

Table 3.5 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

For implementations which implement multiple level of caches and where the hardware maintains the smaller cache as a proper subset of a larger cache (every address which is resident in the smaller cache is also resident in the larger cache; also known as the inclusion property), it is recommended that the CACHE instructions which operate on the larger, outer-level cache; should first operate on the smaller, inner-level cache. For example, a Hit_Writeback _Invalidate operation targeting the Secondary cache, should first operate on the primary data cache first. If the CACHE instruction implementation does not follow this policy then any software which flushes the caches must mimic this behavior. That is, the software sequences must first operate on the inner cache then operate on the outer cache. The software must place a SYNC instruction after the CACHE instruction whenever there are possible write-

backs from the inner cache to ensure that the writeback data is resident in the outer cache before operating on the outer cache. If neither the CACHE instruction implementation nor the software cache flush sequence follow this policy, then the inclusion property of the caches can be broken, which might be a condition that the cache management hardware cannot properly deal with.

For implementations which implement multiple level of caches without the inclusion property, the use of a SYNC instruction after the CACHE instruction is still needed whenever writeback data has to be resident in the next level of memory hierarchy.

For multiprocessor implementations that maintain coherent caches, some of the Hit type of CACHE instruction operations may optionally affect all coherent caches within the implementation. If the effective address uses a coherent Cache Coherency Attribute (CCA), then the operation is *globalized*, meaning it is broadcast to all of the coherent caches within the system. If the effective address does not use one of the coherent CCAs, there is no broadcast of the operation. If multiple levels of caches are to be affected by one CACHE instruction, all of the affected cache levels must be processed in the same manner - either all affected cache levels use the globalized behavior or all affected cache levels use the non-globalized behavior.

Table 3.6 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 power up.	Required
	S, T	Index Writeback Invalidate / Index Invalidate	Index		Required if S, T cache is implemented

Table 3.6 Encoding of Bits [20:18] of the CACHE Instruction (Continued)

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
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
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. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Required (Instruction Cache Encoding Only), Recommended otherwise
	S, T	Hit Invalidate	Address		Optional, if Hit_Invalidate_D is implemented, the S and T variants are recommended.

Table 3.6 Encoding of Bits [20:18] of the CACHE Instruction (Continued)

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. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Required
	S, T	Hit Writeback Invalidate / Hit Invalidate	Address		Required if S, T cache is implemented
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. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Recommended
	S, T	Hit Writeback	Address		Optional, if Hit_Writeback_D is implemented, the S and T variants are recommended.

Table 3.6 Encoding of Bits [20:18] of the CACHE Instruction (Continued)

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 write-back 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 this instruction is used to lock all ways of a cache at a specific cache index, the behavior of that cache to subsequent cache misses to that cache index is **UNDEFINED**.

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

Any use of this instruction that can cause cacheline writebacks should be followed by a subsequent SYNC instruction to avoid hazards where the writeback data is not yet visible at the next level of the memory hierarchy.

This instruction does not produce an exception for a misaligned memory address, since it has no memory access size.

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

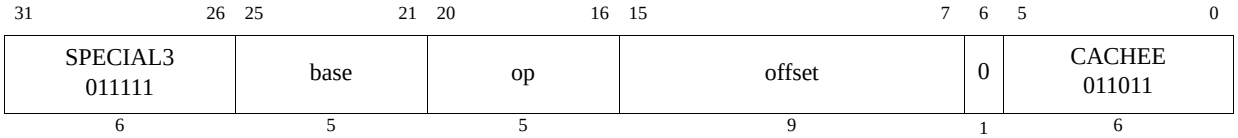
Bus Error Exception

Programming Notes:

As shown in the instruction drawing above, the MIPS Release 6 architecture implements a 9-bit offset, whereas all release levels lower than Release 6 of the MIPS architecture implement a 16-bit offset.

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 an unmapped address (such as an 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 */
```



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

Purpose: Perform Cache Operation EVA

To perform the cache operation specified by op using a user mode virtual address while in kernel mode.

Description:

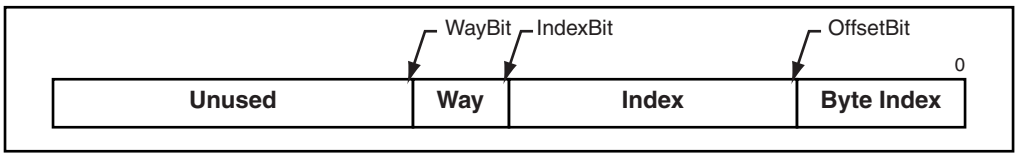
The 9 bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

Table 3.7 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 ← Addr`$_{\texttt{WayBit-1..IndexBit}}$
		`Index ← Addr`$_{\texttt{IndexBit-1..OffsetBit}}$
		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 3.4 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. This instruction never causes Execute-Inhibit nor Read-Inhibit exceptions.

The effective address may be an arbitrarily-aligned by address. The CACHEE 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.

The CACHEE instruction and the memory transactions which are sourced by the CACHEE instruction, such as cache refill or cache writeback, obey the ordering and completion rules of the SYNC instruction.

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

Table 3.8 Encoding of Bits[17:16] of CACHEE 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

For implementations which implement multiple level of caches and where the hardware maintains the smaller cache as a proper subset of a larger cache (every address which is resident in the smaller cache is also resident in the larger cache; also known as the inclusion property), it is recommended that the CACHEE instructions which operate on the larger, outer-level cache; should first operate on the smaller, inner-level cache. For example, a Hit_Writeback _Invalidate operation targeting the Secondary cache, should first operate on the primary data cache first. If the CACHEE instruction implementation does not follow this policy then any software which flushes the caches must mimic this behavior. That is, the software sequences must first operate on the inner cache then operate on the outer cache. The software must place a SYNC instruction after the CACHEE instruction whenever there are possible writebacks from the inner cache to ensure that the writeback data is resident in the outer cache before operating on the outer cache. If neither the CACHEE instruction implementation nor the software cache flush sequence follow this policy, then the inclusion property of the caches can be broken, which might be a condition that the cache management hardware cannot properly deal with.

For implementations which implement multiple level of caches without the inclusion property, the use of a SYNC instruction after the CACHEE instruction is still needed whenever writeback data has to be resident in the next level of memory hierarchy.

For multiprocessor implementations that maintain coherent caches, some of the Hit type of CACHEE instruction operations may optionally affect all coherent caches within the implementation. If the effective address uses a coherent Cache Coherency Attribute (CCA), then the operation is *globalized*, meaning it is broadcast to all of the coherent caches within the system. If the effective address does not use one of the coherent CCAs, there is no broadcast of the operation. If multiple levels of caches are to be affected by one CACHEE instruction, all of the affected cache levels must be processed in the same manner - either all affected cache levels use the globalized behavior or all affected cache levels use the non-globalized behavior.

The CACHEE instruction functions in exactly the same fashion as the CACHE instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible . Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Table 3.9 Encoding of Bits [20:18] of the CACHEE 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 power up.	Required
	S, T	Index Writeback Invalidate / Index Invalidate	Index		Required if S, T cache is implemented

Table 3.9 Encoding of Bits [20:18] of the CACHEE Instruction (Continued)

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
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
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. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Required (Instruction Cache Encoding Only), Recommended otherwise
	S, T	Hit Invalidate	Address		Optional, if Hit_Invalidate_D is implemented, the S and T variants are recommended.

Table 3.9 Encoding of Bits [20:18] of the CACHEE Instruction (Continued)

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. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Required
	S, T	Hit Writeback Invalidate / Hit Invalidate	Address		Required if S, T cache is implemented
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. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Recommended
	S, T	Hit Writeback	Address		Optional, if Hit_Writeback_D is implemented, the S and T variants are recommended.

Table 3.9 Encoding of Bits [20:18] of the CACHEE Instruction (Continued)

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 write-back 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 CACHEE instruction is the target of an invalidate or a writeback invalidate.

If this instruction is used to lock all ways of a cache at a specific cache index, the behavior of that cache to subsequent cache misses to that cache index is **UNDEFINED**.

Any use of this instruction that can cause cacheline writebacks should be followed by a subsequent SYNC instruction to avoid hazards where the writeback data is not yet visible at the next level of the memory hierarchy.

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

This instruction does not produce an exception for a misaligned memory address, since it has no memory access size.

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

Reserved Instruction

Address Error Exception

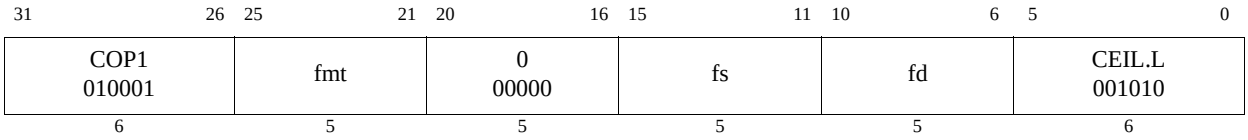
Cache Error Exception

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 */
```



Format: `CEIL.L.fmt`

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

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

Purpose: Fixed Point Ceiling Convert to Long Fixed Point

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, 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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

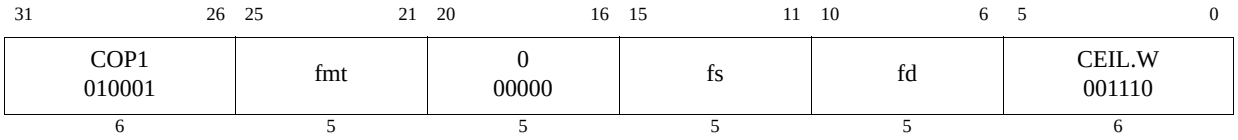
```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact



Format: `CEIL.W.fmt`

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

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

Purpose: Floating Point Ceiling Convert to Word Fixed Point

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:

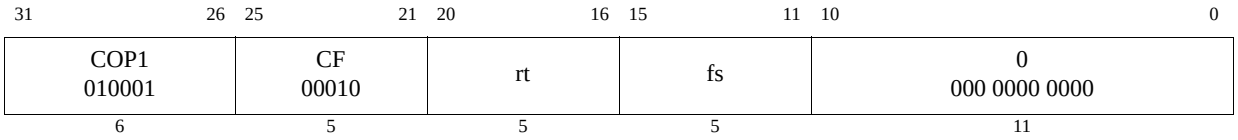
```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact



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

Purpose: Move Control Word From Floating Point

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

Description: `GPR[rt] ← FP_Control[fs]`

Copy the 32-bit word from FP (coprocessor 1) control register *fs* into GPR *rt*, sign-extending it to 64 bits.

UFR (User FR change facility):

The definition of this instruction has been extended in Release 5 to support user mode read of $Status_{FR}$ under the control of $Config5_{UFR}$. This optional feature is meant to facilitate transition from *FR*=0 to *FR*=1 floating-point register modes in order to obsolete *FR*=0 mode. An implementation that is strictly *FR*=1 would not support this feature.

The UFR facility is removed by MIPS32 Release 6. Accessing the UFR and UNFR registers cannot occur because MIPS32 Release 6 does not allow FIR_{UFRP} to be set.

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.

In particular, the result is **UNPREDICTABLE** if *fs* specifies the UNFR write-only control. R5.03 implementations are required to produce a Reserved Instruction Exception; software must assume it is **UNPREDICTABLE.**

Operation:

```
if fs = 0 then
   temp ← FIR
elseif fs = 1 then /* read UFR (CP1 Register 1) */
   if FIR_UFRP then
      if not Config5_UFR then SignalException(RI) endif
      temp := Status_FR
   else
      if Config2.AR>=2 SignalException(RI) /* MIPS Release 6 traps */
      temp := UNPREDICTABLE
   endif
/* note: fs=4 UNFR not supported for reading - UFR suffices */
elseif fs = 25 then /* FCCR */
   temp ← 0^24 || FCSR_31..25 || FCSR_23
elseif fs = 26 then /* FEXR */
   temp ← 0^14 || FCSR_17..12 || 0^5 || FCSR_6..2 || 0^2
elseif fs = 28 then /* FENR */
   temp ← 0^20 || FCSR_11.7 || 0^4 || FCSR_24 || FCSR_1..0
elseif fs = 31 then /* FCSR */
   temp ← FCSR
else
```

```
    temp ← UNPREDICTABLE
endif

GPR[rt] ← sign_extend(temp)
```

Exceptions:

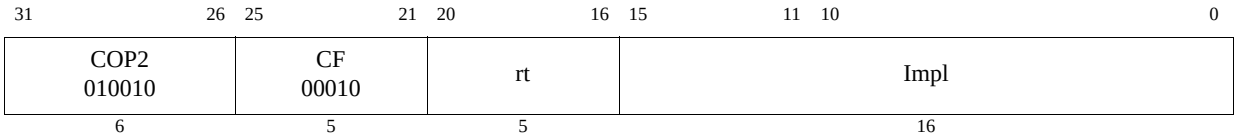
Coprocessor Unusable, Reserved Instruction

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.

MIPS32r5 introduced the UFR and UNFR register aliases that allow user level access to $Status_{FR}$. MIPS32 Release 6 removes them.



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

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

Purpose: Move Control Word From Coprocessor 2

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:

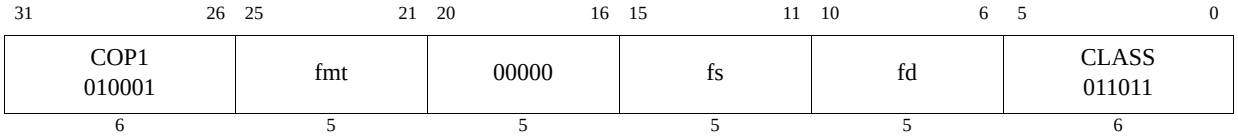
The result is **UNPREDICTABLE** if *Impl* specifies a register that does not exist.

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction



Format: `CLASS.fmt`
`CLASS.S fd,fs` **MIPS32 Release 6**
`CLASS.D fd,fs` **MIPS32 Release 6**

Purpose: Scalar Floating-Point Class Mask

Scalar floating-point class shown as a bit mask for Zero, Negative, Infinite, Subnormal, Quiet NaN, or Signaling NaN.

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

Stores in *fd* a bit mask reflecting the floating-point class of the floating point scalar value *fs*.

The mask has 10 bits as follows. Bits 0 and 1 indicate NaN values: signaling NaN (bit 0) and quiet NaN (bit 1). Bits 2, 3, 4, 5 classify negative values: infinity (bit 2), normal (bit 3), subnormal (bit 4), and zero (bit 5). Bits 6, 7, 8, 9 classify positive values: infinity (bit 6), normal (bit 7), subnormal (bit 8), and zero (bit 9).

This instruction corresponds to the **class** operation of the IEEE Standard for Floating-Point Arithmetic 754™-2008. This scalar FPU instruction also corresponds to the vector FCLASS.df instruction of MSA.

The input values and generated bit masks are not affected by the flush-subnormal-to-zero mode FCSR.FS.

The input operand is a scalar value in floating-point data format *fmt*. Bits beyond the width of *fmt* are ignored. The result is a bitmask as described above, zero extended to coprocessor register width (i.e. 64 bits on an FPU without MSA, 128 bits if the processor supports MSA).

Restrictions:

No data-dependent exceptions are possible.

CLASS.fmt is defined only for formats S and D. Other formats must produce a Reserved Instruction Exception (unless used for a different instruction).

Availability:

Instruction CLASS.fmt is introduced by and required as of MIPS32 Release 6.

Operation:

```
CLASS.fmt
ValidateAccessToFPUResources(fmt,{S,D})
   fin ← ValueFPR(fs,fmt)
   masktmp ← ClassFP(fin, fmt)
   StoreFPR (fd, fmt, ftmp )
   ?

function ClassFP(tt, ts, n)
   /* Implementation defined class operation. */
endfunction ClassFP
```

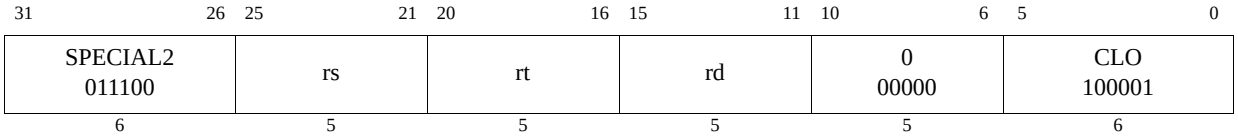
Exceptions:

Coprocessor Unusable, Reserved Instruction

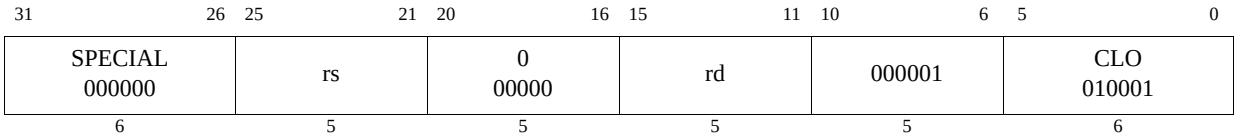
Floating Point Exceptions:

Unimplemented Operation

MIPS32, (all release levels less than Release 6)



MIPS32 Release 6 (Release 6 and future)



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

Purpose: Count Leading Ones in Word

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:

Prior to MIPS32 Release 6: 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. MIPS32 Release 6's new instruction encoding does not contain an *rt* field.

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
```

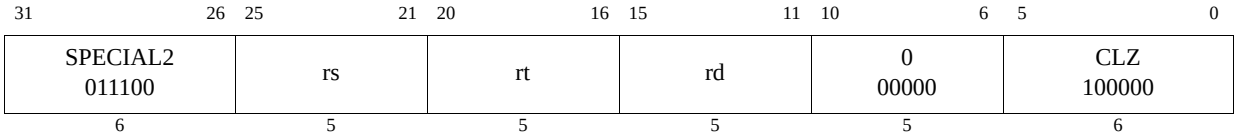
Exceptions:

None

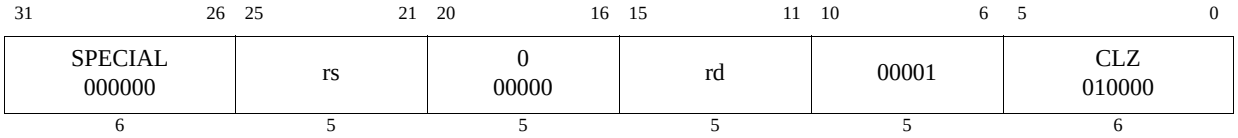
Programming Notes:

As shown in the instruction drawing above, the MIPS Release 6 architecture sets the 'rt' field to a value of 00000.

MIPS32, (all release levels less than Release 6)



MIPS32 Release 6 (Release 6 and future)



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

Purpose: Count Leading Zeros in Word

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 *rd*is 32.

Restrictions:

pre-MIPS32 Release 6: 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. MIPS32 Release 6's new instruction encoding does not contain an *rt* field.

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:

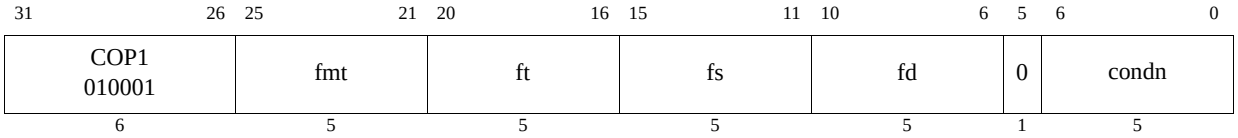
```
if NotWordValue(GPR[rs]) then
   UNPREDICTABLE
endif
temp ← 32
for i in 31 .. 0
   if GPR[rs]i = 1 then
      temp ← 31 - i
      break
   endif
endfor
GPR[rd] ← temp
```

Exceptions:

None

Programming Notes:

As shown in the instruction drawing above, the MIPS Release 6 architecture sets the 'rt' field to a value of 00000.



Format: `CMP.condn.fmt`

`CMP.condn.S fd, fs, ft (with fmt=10100 (W))` **MIPS32 Release 6**

`CMP.condn.D fd, fs, ft (with fmt=10101 (L))` **MIPS32 Release 6**

Purpose: Floating Point Compare setting Mask

To compare FP values and record the result as a mask of all 0s or all 1s in a floating point register

Description: `FPR[fd] ← Extend.fmt(FPR[fs] compare_cond FPR[ft]`

The value in FPR *fs* is compared to the value in FPR *ft*.

The comparison is exact and neither overflows nor underflows.

If the comparison specified by the *condn* field of the instruction 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 FPR *fd*; true is all 1s and false is all 0s, repeated the operand width of *fmt*. All other bits beyond the operand width *fmt* are UNPREDICTABLE. E.g. a 32-bit single precision comparison writes a mask of 32 0s or 1s into bits 0 to 31 of FPR *fd*, and makes bits 32 to 63 to UNPREDICTABLEif a 64-but FPU without MSA, or bits 32 to 127 if MSA is present.

The values are in format *fmt*. However, these instructions use a non-standard encoding of *fmt: fmt* encoding=10100, which is W (32-bit integer) elsewhere, means S (32-bit single precision floating point) here; f*mt* encoding=10101, which is L (64-bit integer) elsewhere, means D (64-bit double precision floating point) here.

All other encodings, i.e. all other values of *fmt,* are reserved in MIPS32 Release 6, and produce a Reserved Instruction Exception. In particular, the encodings corresponding to MIPS32 Release 5 C.cond.S and C.cond.D are so reserved.

The *condn* field of the instruction specifies the nature of the comparison: equals, less than, and so on, unordered or ordered, signalling or quiet, as specified in Table 3.10 “Comparing CMP.condn.fmt, IEEE 754-2008, C.cond.fmt, and MSA FP compares” on page 194.

In this release of MIPS32 Release 6, it can be seen that the *condn* field bits have specific purposes: $cond_4$, and $cond_{2..1}$ specify the nature of the comparison (equals, less than, and so on); $cond_0$ specifies whether the comparison is ordered or unordered, i.e. false or true if any operand is a NaN; $cond_3$ indicates whether the instruction should signal an exception on QNaN inputs. However, in the future the MIPS ISA may be extended in ways that do not preserve these meanings.

All encodings of the *condn* field that are not specified, e.g. which are shaded in Table 3.10, are reserved in MIPS32 Release 6 and produce a Reserved Instruction Exception.

If one of the values is an SNaN, or if a signalling comparison is specified (currently, $cond_3$ 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 mask result is written into FPR *fd.*

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.

The predicates implemented are described in Table 3.10 “Comparing CMP.condn.fmt, IEEE 754-2008, C.cond.fmt, and MSA FP compares” on page 194. Not all of the 16 IEEE predicates are implemented directly by hardware. For the directed comparisons (LT, LE, GT, GE) the missing predicates can be obtained by reversing the FPR register operands *ft* and *fs*. E.g. the hardware implements the “Ordered Less Than” predicate LT(fs,ft); reversing the operands LT(ft,fs) produces the dual predicate “Unordered or Greater Than or Equal” UGE(fs,ft). Table 3.10 shows these mappings. Reversing inputs is ineffective for the symmetric predicates such as EQ; MIPS32 Release 6 implements these negative predicates directly, so that all mask values can be generated in a single instruction.

In addition to showing the comparison predicates, instruction encodings, instruction mnemonics and longer instruction names for CMP.condn.fmt, Table 3.10 provides comparisons to (1) the MIPS32 Release 5 C.cond.fmt,, and (2) the (MSA) MIPS SIMD Architecture packed vector floating point comparison instructions. CMP.condn.fmt provides exactly the same comparisons for FPU scalar values that MSA provides for packed vectors, with similar mnemonics. CMP.condn.fmt provides a superset of the MIPS32 Release 5 C.cond.fmt comparisons.

The official MIPS32 Release 6 mnemonics for comparisons have slightly changed: e.g. SULT, Signalling Unordered or Less Than. Table 3.10 indicates mnemonics that implement the same predicates.

In addition, Table 3.10 shows the corresponding IEEE 754-2008 comparison operations.

Table 3.10 Comparing CMP.condn.fmt, IEEE 754-2008, C.cond.fmt, and MSA FP compares

Shaded entries in the table are unimplemented, and reserved.

Instruction Encodings

```
CMP.condn.fmt: 010001 fffff ttttt sssss ddddd 0ccccc
  C.cond.fmt: 010001 fffff ttttt sssss CCC00 11cccc
         MSA: 011110 oooof ttttt sssss ddddd mmmmmm
```

Invalid Operand Exception		**MSA: operation** oooo *Bits 25…22* **C: cond** cccc - Bits 3..0 **CMP: condn** cccccc - *Bits 3..0*	**MSA: minor opcode** mmmmmm *Bits 5…0* = 26 - 011010 **CMP: condn** Bit 5..4 = 00 **C**: only applicable									**MSA: minor opcode** mmmmmm *Bits 5…0* = 28 - 011100 **CMP: condn** Bit 5..4 = 01 **C**: not applicable							
			Predicates									**Negated Predicates**							
			Relation				C condn.fmt	MSA	CMP condn.fmt	**Long names**	IEEE	Relation				C condn.fmt	MSA	CMP condn.fmt	**Long names**	IEEE
			>	<	=	?						>	<	=	?					
no (non-signalling)	yes (always signal SNaN)	0 0000	F	F	F	F	F	FCAF	AF	False Always False		T	T	T	T	T		AT	True Always True	
		1 0001	F	F	F	T	UN	FCUN	UN	Unordered	compareQuietUnordered ? isUnordered	T	T	T	F	OR	FCOR	OR	Ordered	compareQuietOrdered <=> NOT(isUnordered)
		2 0010	F	F	T	F	EQ	FCEQ	EQ	Equal	compareQuietEqual =	T	T	F	T	NEQ	FCUNE	UNE	Not Equal	compareQuietNotEqual ?<>, NOT(=), ≠
		3 0011	F	F	T	T	UEQ	FCUEQ	UEQ	Unordered or Equal		T	T	F	F	OGL	FCNE	NE	Ordered Greater Than or Less Than	
		4 0100	F	T	F	F	OLT	FCLT	LT	Ordered Less Than	compareQuietLess isLess	T	F	T	T	UGE		UGE	Unordered or Greater Than or Equal	compareQuietNotLess ?>=, NOT(isLess)
		5 0101	F	T	F	T	ULT	FCULT	ULT	Unordered or Less Than	compareQuietLessUnor- dered ?<, NOT(isGreaterEqual)	T	F	T	F	OGE		OGE	Ordered Greater Than or Equal	compareQuiet- GreatrEqual isGreaterEqual
		6 0110	F	T	T	F	OLE	FCLE	LE	Ordered Less than or Equal	compareQuietLessEqual isLessEqual	T	F	F	T	UGT		UGT	Unordered or Greater Than	compareQuietGreaterUn- ordered ?>, NOT(isLessEqual)
		7 0111	F	T	T	T	ULE	FCULE	ULE	Unordered or Less Than or Equal	compareQuietNotGreater ?<=, NOT(isGreater)	T	F	F	F	OGT		OGT	Ordered Greater Than	compareQuietGreater isGreater

Table 3.10 Comparing CMP.condn.fmt, IEEE 754-2008, C.cond.fmt, and MSA FP compares (Continued)

Shaded entries in the table are unimplemented, and reserved.

Instruction Encodings

```
CMP.condn.fmt: 010001 fffff ttttt sssss ddddd 0ccccc
  C.cond.fmt: 010001 fffff ttttt sssss CCC00 11cccc
         MSA: 011110 oooof ttttt sssss ddddd mmmmmm
```

Invalid Operand Exception	MSA: operation oooo *Bits 25…22* C: cond cccc - Bits 3..0 CMP: condn cccccc - *Bits 3..0*	MSA: minor opcode mmmmmm *Bits 5…0* = 26 - 011010 CMP: condn Bit 5..4 = 00 C: only applicable — Predicates									MSA: minor opcode mmmmmm *Bits 5…0* = 28 - 011100 CMP: condn Bit 5..4 = 01 C: not applicable — Negated Predicates								
		Relation >	<	=	?	C condn.fmt	MSA	CMP condn.fmt	Long names	IEEE	Relation >	<	=	?	C condn.fmt	MSA	CMP condn.fmt	Long names	IEEE
yes (signalling)	8 1000	F	F	F	F	SF	FSAF	SAF	Signalling False Signalling Always False		T	T	T	T	ST		SAT	Signalling True Signalling Always True	
	9 1001	F	F	F	T	NGLE	FSUN	SUN	Not Greater Than or Less Than or Equal Signalling Unordered		T	T	T	F	GLE	FSOR	SOR	Greater Than or Less Than or Equal Signalling Ordered	
	10 1010	F	F	T	F	SEQ	FSEQ	SEQ	Signalling Equal Ordered Signalling Equal	compareSignalling Equal	T	T	F	T	SNE	FSUNE	SUNE	Signalling Not Equal Signalling Unordered or Not Equal	compareSignalling-NotEqual
	11 1011	F	F	T	T	NGL	FSUEQ	SUEQ	Not Greater Than or Less Than Signalling Unordered or Equal		T	T	F	F	GL	FSNE	SNE	Greater Than or Less Than Signalling Ordered Not Equal	
	12 1100	F	T	F	F	LT	FSLT	SLT	Less Than Ordered Signalling Less Than	compareSignallingLess <	T	F	T	T	NLT		SUGE	Not Less Than Signalling Unordered or Greater Than or Equal	compareSignallingNot-Less NOT(<)
	13 1101	F	T	F	T	NGE	FSULT	SULT	Not Greater Than or Equal Unordered or Less Than	compareSignalling-LessUnordered NOT(>=)	T	F	T	F	GE		SOGE	Signalling Ordered Greater Than or Equal	compareSignalling-GreaterEqual >=, ≥
	14 1110	F	T	T	F	LE	FSLE	SLE	Less Than or Equal Ordered Signalling Less Than or Equal	compareSignalling-LessEqual <=, ≤	T	F	F	T	NLE		SUGT	Not Less Than or Equal Signalling Unordered or Greater Than	compareSignalling-GreaterUnordered NOT(<=)
	15 1111	F	T	T	T	NGT	FSULE	SULE	Not Greater Than Signalling Unordered or Less Than or Equal	compareSignalling-NotGreater NOT(>)	T	F	F	F	GT		SOGT	Greater Than Signalling Ordered Greater Than	compareSignalling-Greater >

Restrictions:

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 ← cond4 xor (
          (cond2 and less)
          or (cond1 and equal)
          or (cond0 and unordered) )
   StoreFPR (fd, fmt, ExtendBit.fmt(condition))
endif
```

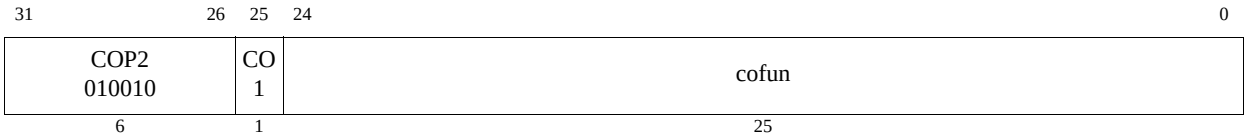
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

DocumentMarginNote



Format: `COP2 func` **MIPS32**

Purpose: Coprocessor Operation to Coprocessor 2

To perform an operation to Coprocessor 2

Description: `CoprocessorOperation(2, cofun)`

An implementation-dependent operation is performed 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.

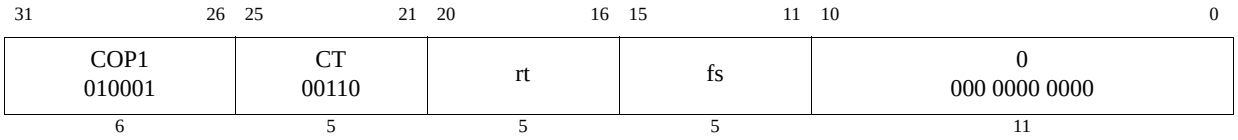
Restrictions:

Operation:

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

Exceptions:

Coprocessor Unusable
Reserved Instruction



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

Purpose: Move Control Word to Floating Point

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.

UFR (User FR change facility):

The definition of this instruction has been extended in Release 5 to support user mode read of $Status_{FR}$ under the control of $Config5_{UFR}$. This optional feature is meant to facilitate transition from *FR*=0 to *FR*=1 floating-point register modes in order to obsolete *FR*=0 mode. An implementation that is strictly *FR*=1 would not support this feature.

The UFR facility is removed by MIPS32 Release 6. Accessing the UFR and UNFR registers cannot occur because MIPS32 Release 6 does not allow FIR_{UFRP} to be set.

Preliminary

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.

Furthermore, the result is **UNPREDICTABLE** if *fd* specifies the UFR or UNFR aliases, with *fs* anything other than 00000, GPR[0]. R5.03 implementations are required to produce a Reserved Instruction Exception; software must assume it is **UNPREDICTABLE.**

Operation:

```
temp ← GPR[rt]31..0
if (fs = 1 or fs = 4) then
   /* clear UFR or UNFR(CP1 Register 1)*/
   if Config2.AR>=2 SignalException(RI) /* MIPS Release 6 traps */
   if not (rt = 0 and FIRUFRP) then UNPREDICTABLE /*end of instruction*/ endif
   if not Config5UFR then signalException(RI) endif
   if fs = 1 then StatusFR ← 0
   elseif fs = 4 then StatusFR ← 1
   else /* cannot happen */
elseif fs = 25 then /* FCCR */
   if temp31..8 ≠ 024 then
      UNPREDICTABLE
   else
```

```
        FCSR ← temp7..1 || FCSR24 || temp0 || FCSR22..0
    endif
elseif fs = 26 then /* FEXR */
    if temp31..18 ≠ 0 or temp11..7 ≠ 0 or temp2..0 ≠ 0then
        UNPREDICTABLE
    else
        FCSR ← FCSR31..18 || temp17..12 || FCSR11..7 ||
        temp6..2 || FCSR1..0
    endif
elseif fs = 28 then /* FENR */
    if temp31..12 ≠ 0 or temp6..3 ≠ 0 then
        UNPREDICTABLE
    else
        FCSR ← FCSR31..25 || temp2 || FCSR23..12 || temp11..7
        || FCSR6..2 || temp1..0
    endif
elseif fs = 31 then /* FCSR */
    if (FCSRImpl field is not implemented) and(temp22..18 ≠ 0) then
        UNPREDICTABLE
    elseif (FCSRImpl field is implemented) and temp20..18 ≠ 0 then
        UNPREDICTABLE
    else
        FCSR ← temp
    endif
else
    UNPREDICTABLE
endif

CheckFPException()
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

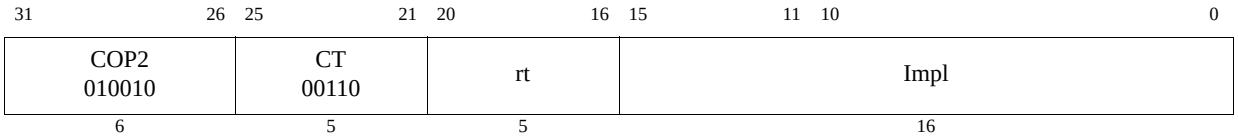
Unimplemented Operation, Invalid Operation, Division-by-zero, Inexact, Overflow, Underflow

Historical Information:

For the MIPS I, II and III architectures, the contents of floating point control register *fs* are **UNPREDICTABLE** 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.

MIPS32 Release 5 introduced the UFR and UNFR register aliases that allow user level access to $Status_{FR}$.



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

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

Purpose: Move Control Word to Coprocessor 2

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:

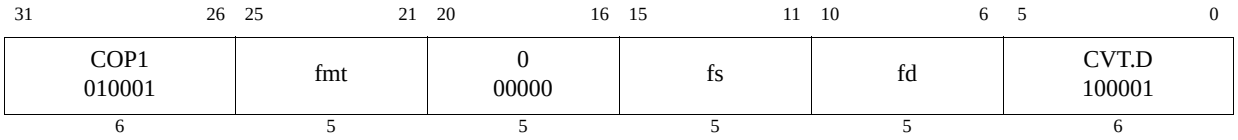
The result is **UNPREDICTABLE** if *rd* specifies a register that does not exist.

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction



Format: `CVT.D.fmt`

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

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

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

Purpose: Floating Point Convert to Double Floating Point

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 the *FR*=0 32-bit FPU register model; i.e. it is the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

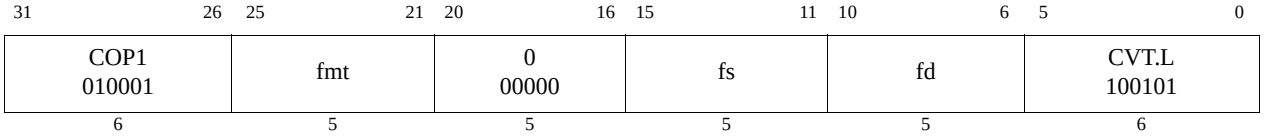
```
StoreFPR (fd, D, ConvertFmt(ValueFPR(fs, fmt), fmt, D))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact



Format: `CVT.L.fmt`

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

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

Purpose: Floating Point Convert to Long Fixed Point

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

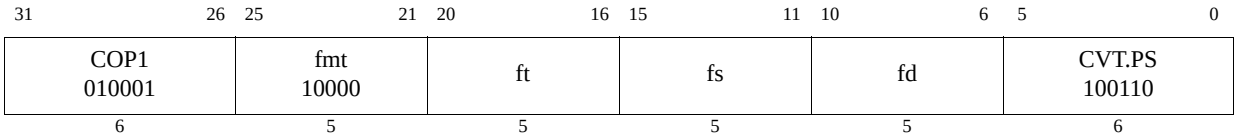
```
StoreFPR (fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact,



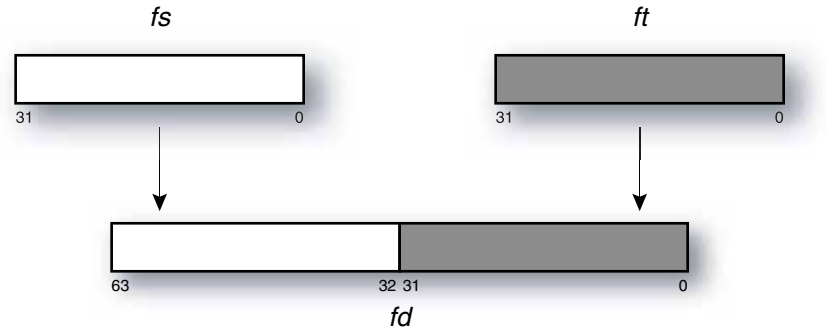
Format: `CVT.PS.S fd, fs, ft` **MIPS64, MIPS32 Release 2,**

Purpose: Floating Point Convert Pair to Paired Single

To convert two FP values to a paired single value

Description: `FPR[fd] ← FPR[fs]31..0 || 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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

This instruction has been removed in the Release 6 architecture.

Operation:

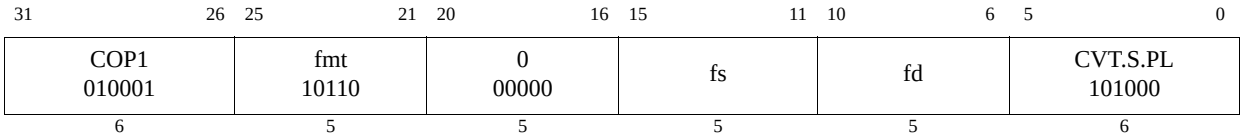
```
StoreFPR(fd, S, ValueFPR(fs,S) || ValueFPR(ft,S))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation



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

Purpose:

Floating Point Convert Pair Lower to Single Floating Point

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

Description: `FPR[fd] ← FPR[fs]`$_{31..0}$

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

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

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

This instruction has been removed in the Release 6 architecture.

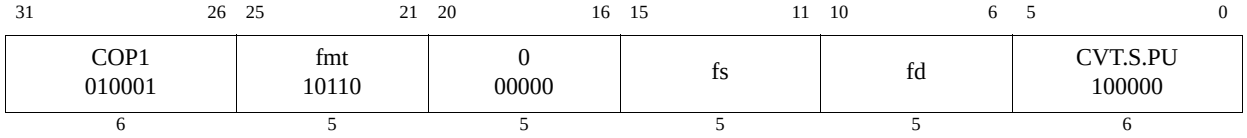
Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:



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

Purpose: Floating Point Convert Pair Upper to Single Floating Point

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

Description: `FPR[fd] ← FPR[fs]`$_{63..32}$

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

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

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 ithe *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU

Availability:

This instruction has been removed in the Release 6 architecture.

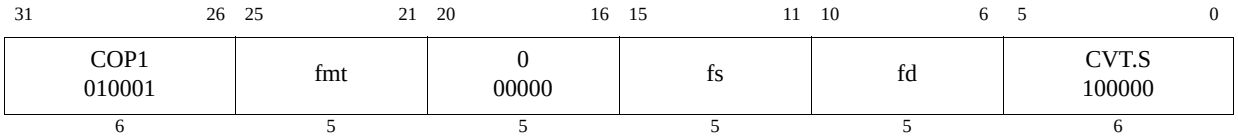
Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:



Format: `CVT.S.fmt`

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

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

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

Purpose: Floating Point Convert to Single Floating Point

To convert an FP or fixed point value to single FP

Description: `FPR[fd] ← convert_and_round(FPR[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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

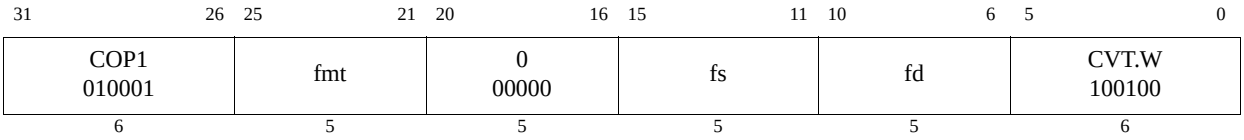
```
StoreFPR(fd, S, ConvertFmt(ValueFPR(fs, fmt), fmt, S))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow, Underflow



Format: `CVT.W.fmt`

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

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

Purpose: Floating Point Convert to Word Fixed Point

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:

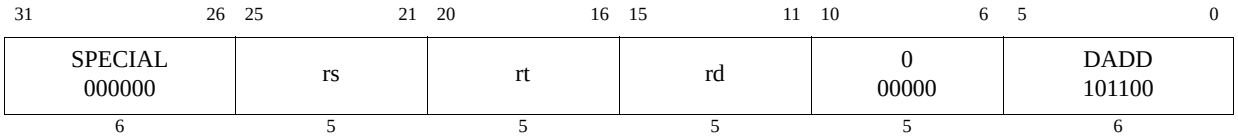
```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact



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

Purpose: Doubleword Add

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:

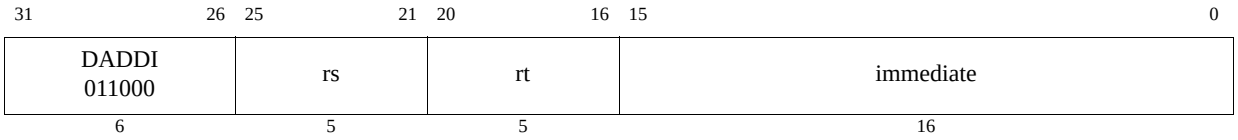
```
temp ← (GPR[rs]63||GPR[rs]) + (GPR[rt]63||GPR[rt])
if (temp64 ≠ temp63) then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp63..0
endif
```

Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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



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

Purpose: Doubleword Add Immediate

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:

Availability:

This instruction is removed by MIPS64 Release 6. The encoding is reused for other instructions introduced by MIPS64 Release 6.

Operation:

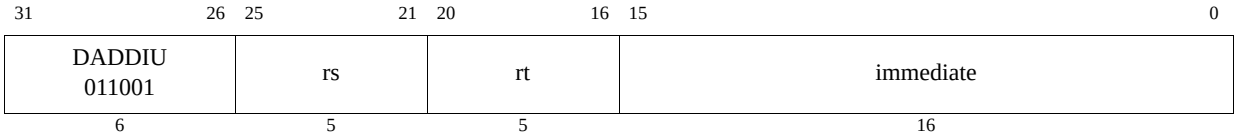
```
temp ← (GPR[rs]63||GPR[rs]) + sign_extend(immediate)
if (temp64 ≠ temp63) then
   SignalException(IntegerOverflow)
else
   GPR[rt] ← temp63..0
endif
```

Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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



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

Purpose: Doubleword Add Immediate Unsigned

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:

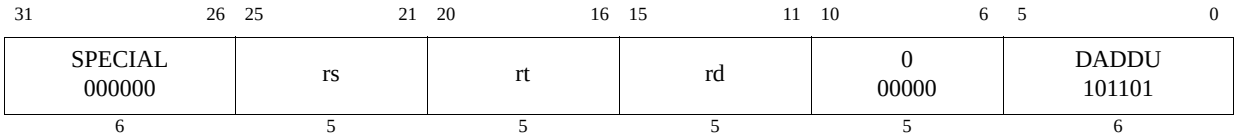
```
GPR[rt] ← GPR[rs] + sign_extend(immediate)
```

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.



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

Purpose: Doubleword Add Unsigned

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]
```

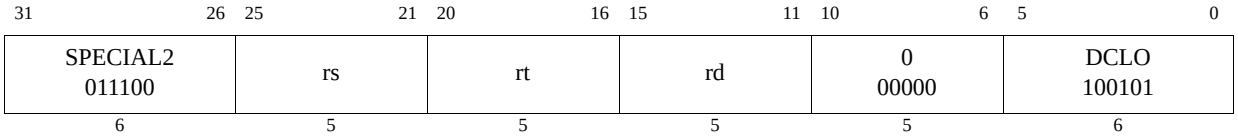
Exceptions:

Reserved Instruction

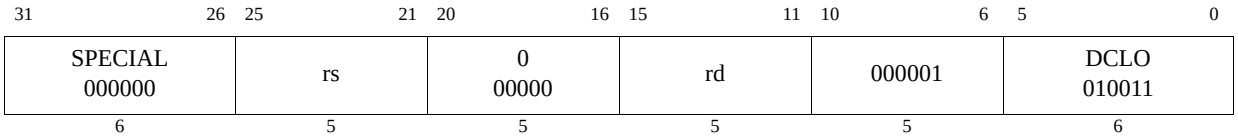
Programming Notes:

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.

MIPS64, (all release levels less than Release 6)



MIPS64 (Release 6 and future)



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

Purpose: Count Leading Ones in Doubleword

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:

Prior to MIPS32 Release 6: 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. MIPS32 Release 6's new instruction encoding does not contain an *rt* field.

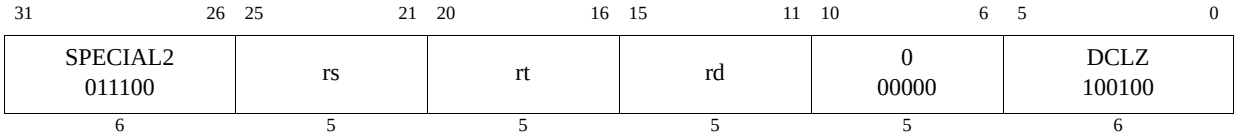
Operation:

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

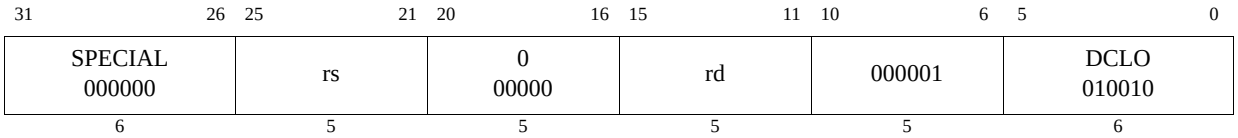
Exceptions:

None

MIPS64, (all release levels less than Release 6)



MIPS64 Release 6 (Release 6 and future)



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

Purpose: Count Leading Zeros in Doubleword

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:

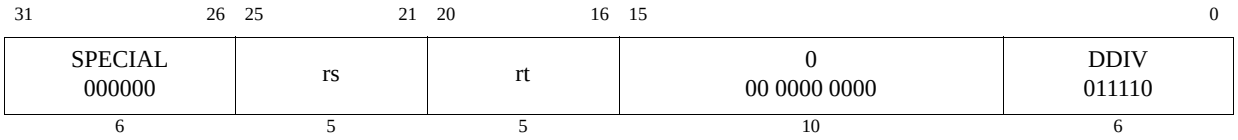
Prior to MIPS32 Release 6: To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the *rt* and *rd* fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values. MIPS32 Release 6's new instruction encoding does not contain an *rt* field.

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

Purpose: Doubleword Divide

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**.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

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

Exceptions:

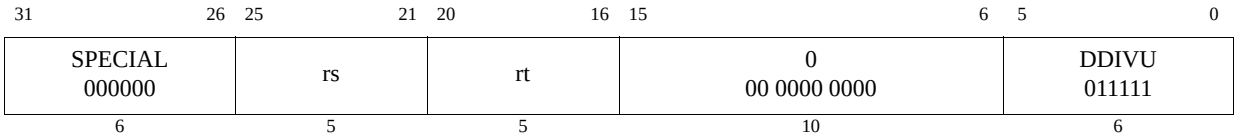
Reserved Instruction

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.



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

Purpose: Doubleword Divide Unsigned

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 **UNPREDICTABLE**.

Availability:

This instruction has been removed in the MIPS64 Release 6 architecture.

Operation:

```
q ← (0 || GPR[rs]) div (0 || GPR[rt])
r ← (0 || GPR[rs]) mod (0 || GPR[rt])
LO ← q63..0
HI ← r63..0
```

Exceptions:

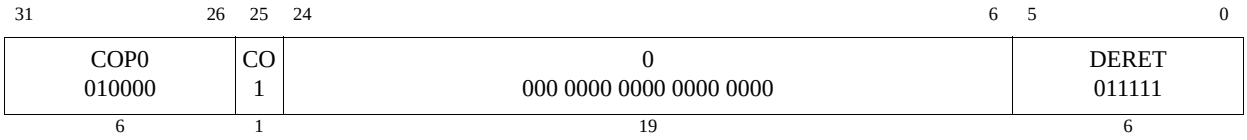
Reserved Instruction

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.



Format: `DERET` **EJTAG**

Purpose: Debug Exception Return

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.pre-MIPS32 Release 6: The operation of the processor is **UNDEFINED** if a DERET is executed in the delay slot of a branch or jump instruction.

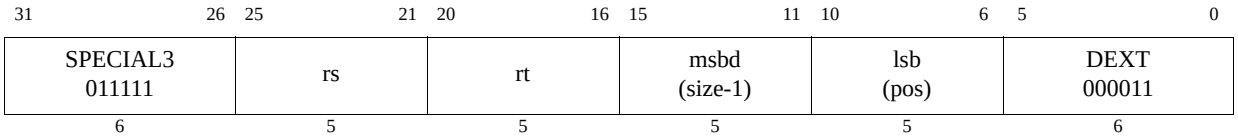
MIPS32 Release 6 implementations are required to signal a Reserved Instruction Exception if DERET is encountered in the delay slot or forbidden slot of a branch or jump instruction.

Operation:

```
Debug_DM ← 0
Debug_IEXI ← 0
if IsMIPS16Implemented() | (Config3_ISA > 0) then
    PC ← DEPC_63..1 || 0
    ISAMode ← DEPC_0
else
    PC ← DEPC
endif
ClearHazards()
```

Exceptions:

Coprocessor Unusable Exception
Reserved Instruction Exception



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

Purpose: Doubleword Extract Bit Field

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:

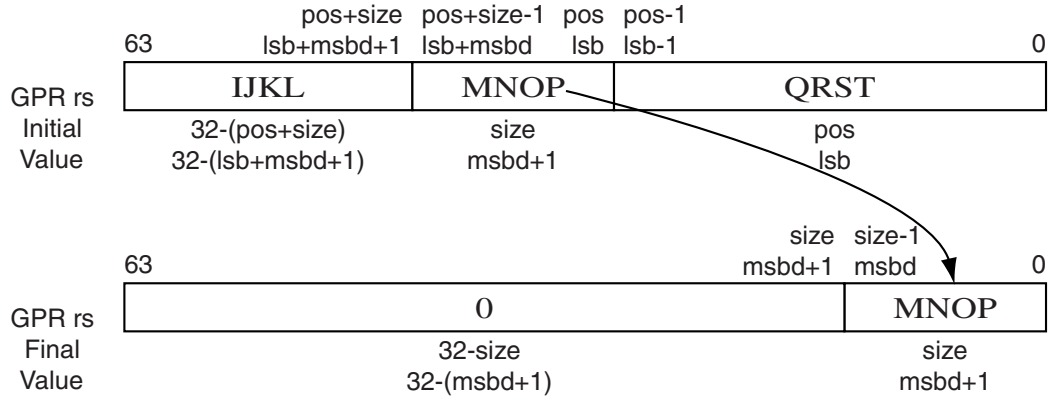
```
msbd ← size-1
lsb ← pos
msb ← lsb+msbd
```

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 3.5 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 ≤ *msbd* < 32	0 ≤ *lsb* < 32	0 ≤ *msb* < 63	0 ≤ *pos* < 32	1 ≤ *size* ≤ 32	DEXT	The field is 32 bits or less and starts in the right-most word of the doubleword
0 ≤ *msbd* < 32	32 ≤ *lsb* < 64	32 ≤ *msb* < 64	32 ≤ *pos* < 64	1 ≤ *size* ≤ 32	DEXTU	The field is 32 bits or less and starts in the left-most word of the doubleword

msbd	lsb	msb	pos	size	Instruction	Comment
$32 \le msbd < 64$	$0 \le lsb < 32$	$32 \le msb < 64$	$0 \le pos < 32$	$32 < size \le 64$	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:

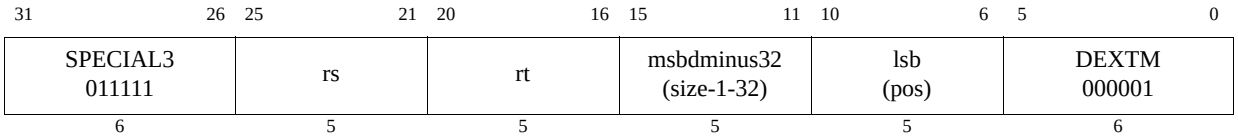
```
GPR[rt] ← 0^(63-(msbd+1)) || GPR[rs]_(msbd+lsb..lsb)
```

Exceptions:

Reserved Instruction

Programming Notes

The assembler will accept any value of *pos* and *size* that satisfies the relationship $0 < pos+size \le 64$ 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: `DEXTM rt, rs, pos, size` **MIPS64 Release 2**

Purpose: Doubleword Extract Bit Field Middle

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:

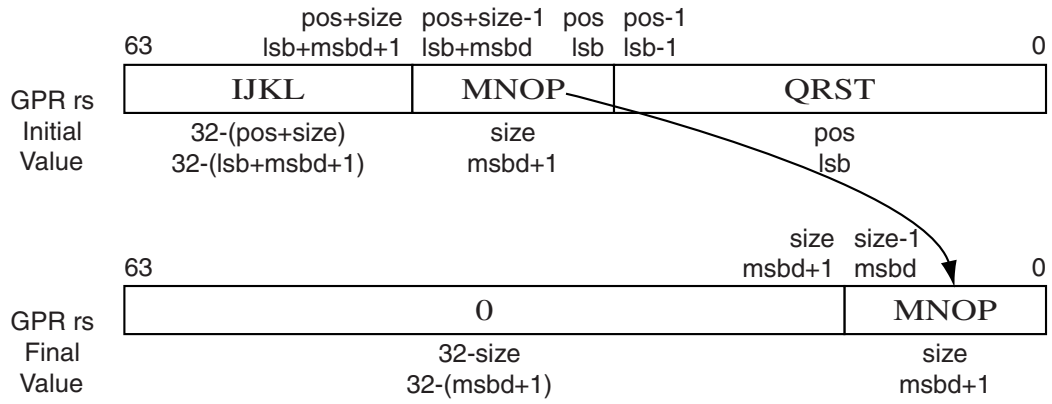
```
msbdminus32 ← size-1-32
lsb ← pos
msbd ← msbdminus32 + 32
msb ← lsb+msbd
```

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 3.6 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 ≤ *msbd* < 32	0 ≤ *lsb* < 32	0 ≤ *msb* < 63	0 ≤ *pos* < 32	1 ≤ *size* ≤ 32	DEXT	The field is 32 bits or less and starts in the right-most word of the doubleword
0 ≤ *msbd* < 32	32 ≤ *lsb* < 64	32 ≤ *msb* < 64	32 ≤ *pos* < 64	1 ≤ *size* ≤ 32	DEXTU	The field is 32 bits or less and starts in the left-most word of the doubleword

msbd	lsb	msb	pos	size	Instruction	Comment
$32 \le msbd < 64$	$0 \le lsb < 32$	$32 \le msb < 64$	$0 \le pos < 32$	$32 < size \le 64$	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:

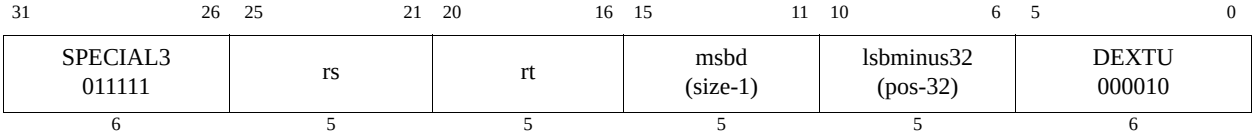
```
msbd ← msbdminus32 + 32
if ((lsb + msbd + 1) > 64) then
   UNPREDICTABLE
endif
GPR[rt] ← 0^(63-(msbd+1)) || GPR[rs]_(msbd+lsb..pos)
```

Exceptions:

Reserved Instruction

Programming Notes

The assembler will accept any value of *pos* and *size* that satisfies the relationship $0 < pos+size \le 64$ 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: Doubleword Extract Bit Field Upper

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:

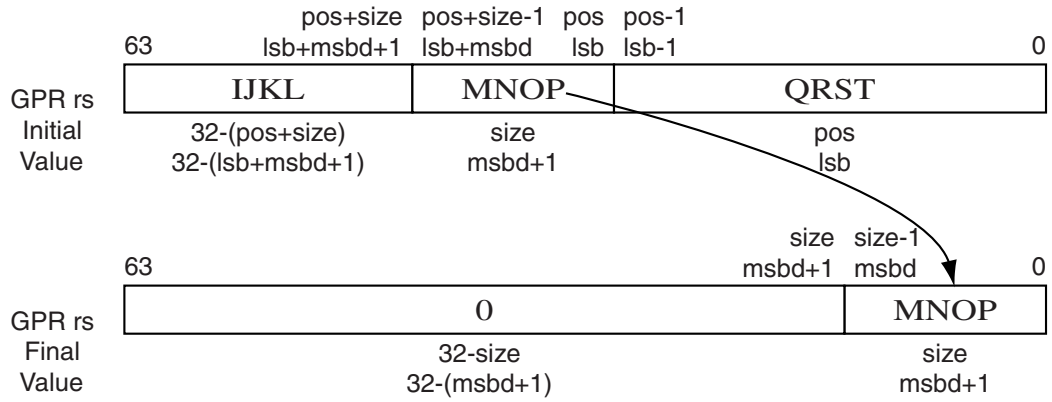
```
msbd ← size-1
lsbminus32 ← pos-32
lsb ← lsbminus32 + 32
msb ← lsb+msbd
```

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 3.7 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 ≤ *msbd* < 32	0 ≤ *lsb* < 32	0 ≤ *msb* < 63	0 ≤ *pos* < 32	1 ≤ *size* ≤ 32	DEXT	The field is 32 bits or less and starts in the right-most word of the doubleword
0 ≤ *msbd* < 32	32 ≤ *lsb* < 64	32 ≤ *msb* < 64	32 ≤ *pos* < 64	1 ≤ *size* ≤ 32	DEXTU	The field is 32 bits or less and starts in the left-most word of the doubleword

msbd	lsb	msb	pos	size	Instruction	Comment
32 ≤ *msbd* < 64	0 ≤ *lsb* < 32	32 ≤ *msb* < 64	0 ≤ *pos* < 32	32 < *size* ≤ 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:

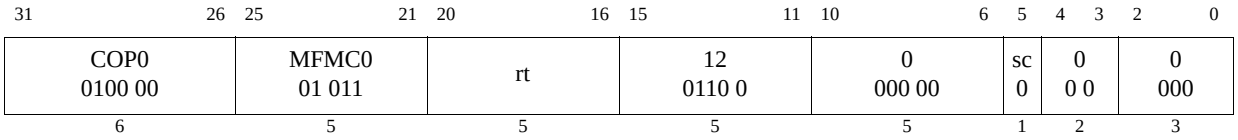
```
lsb ← lsbminus32 + 32
if ((lsb + msbd + 1) > 64) then
   UNPREDICTABLE
endif
GPR[rt] ← 0^(63-(msbd+1)) || GPR[rs]_(msbd+lsb..pos)
```

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.



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

`DI rt` **MIPS32 Release 2**

Purpose: Disable Interrupts

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
```

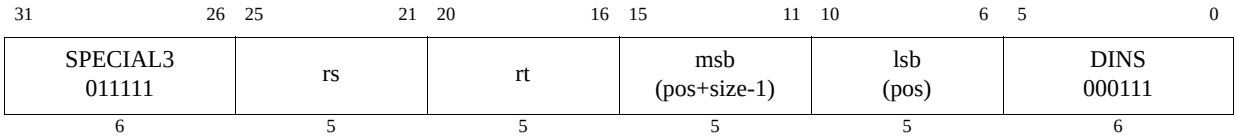
Exceptions:

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 cannot 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: Doubleword Insert Bit Field

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:

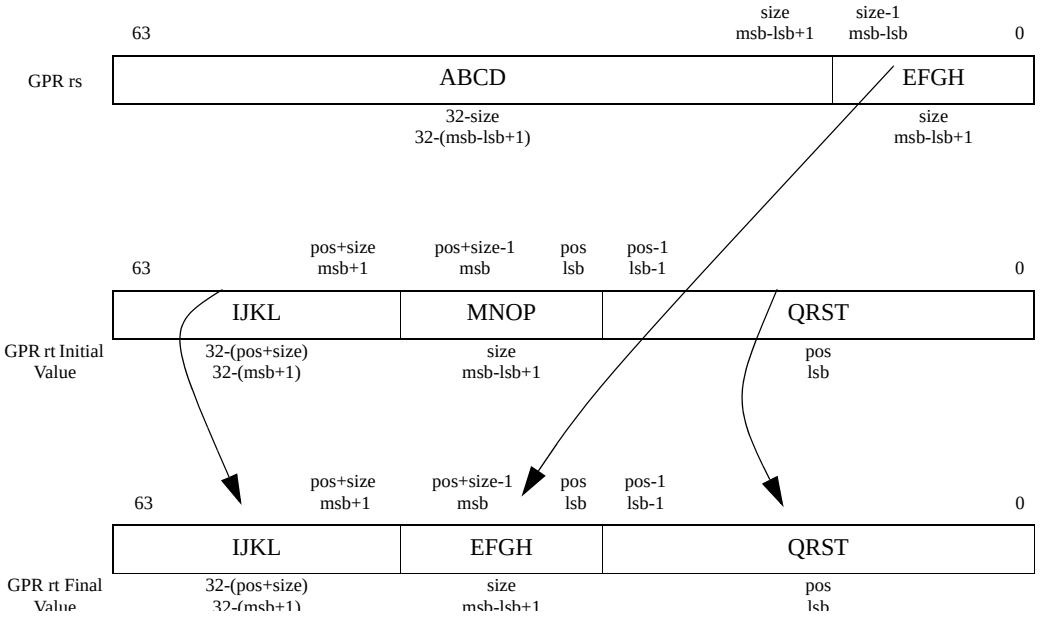
```
msb ← pos+size-1
lsb ← pos
```

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 3.8 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:

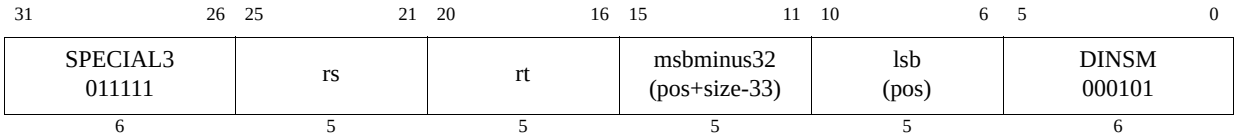
```
if (lsb > msb) then
       UNPREDICTABLE
   endif
   GPR[rt] ← GPR[rt]63..msb+1 || GPR[rs]msb-lsb..0 || GPR[rt]lsb-1..0
```

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.



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

Purpose: Doubleword Insert Bit Field Middle

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:

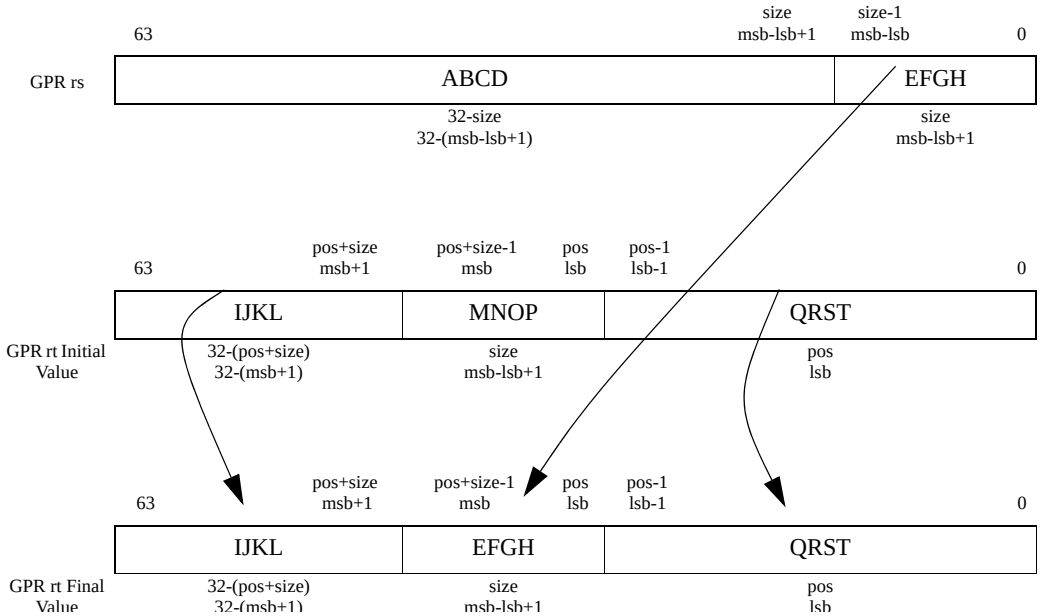
```
msbminus32 ← pos+size-1-32
lsb ← pos
msb ← msbminus32 + 32
```

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 3.9 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:

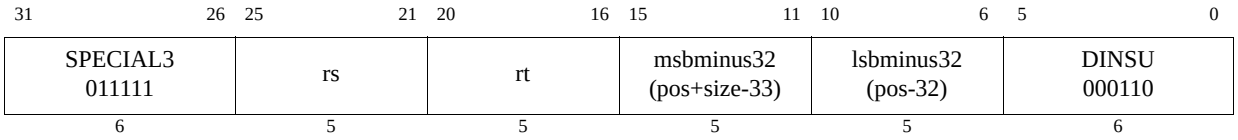
```
msb ← msbminus32 + 32
GPR[rt] ← GPR[rt]63..msb+1 || GPR[rs]msb-lsb..0 || GPR[rt]lsb-1..0
```

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: Doubleword Insert Bit Field Upper

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:

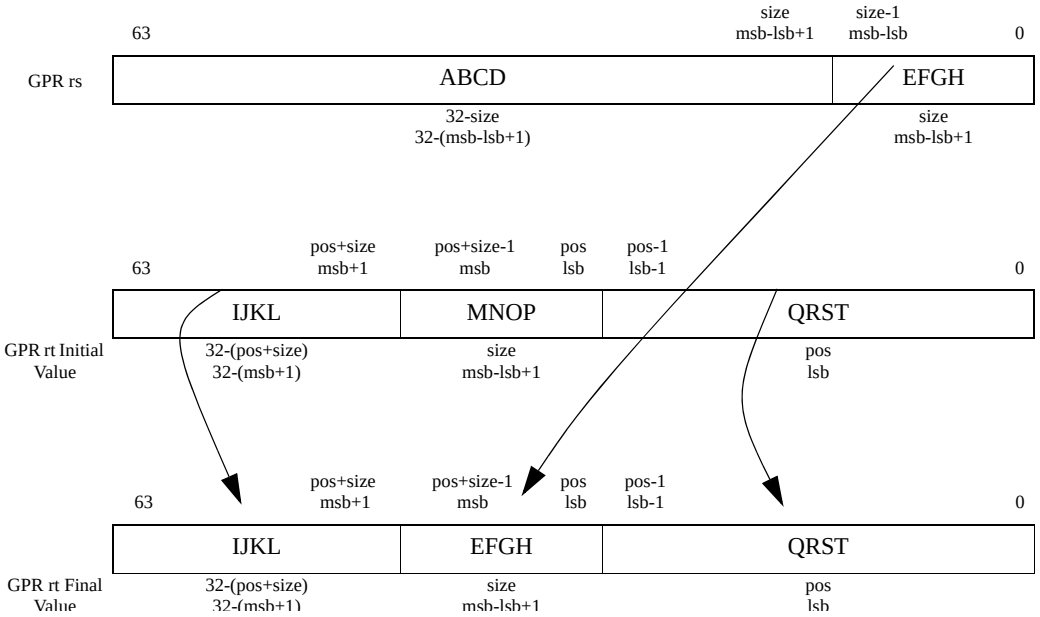
```
msbminus32 ← pos+size-1-32
lsbminus32 ← pos-32
msb ← msbminus32 + 32
lsb ← lsbminus32 + 32
```

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 3.10 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:

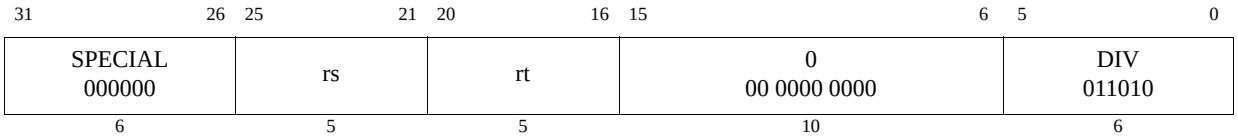
```
lsb ← lsbminus32 + 32
msb ← msbminus32 + 32
if (lsb > msb) then
   UNPREDICTABLE
endif
GPR[rt] ← GPR[rt]63..msb+1 || GPR[rs]msb-lsb..0 || GPR[rt]lsb-1..0
```

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.



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

Purpose: Divide Word

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**.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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:

This pre-MIPS32 Release 6 instruction, DIV, which produces both quotient and remainder, has been removed[1] by MIPS32 Release 6 and has been replaced by DIV and MOD instructions that produce only quotient and remainder, respectively. Refer to the Release 6 introduced ‘DIV’ and ‘MOD’ instructions in this manual for more information. Note that this instruction remains current for all release levels lower than Release 6 of the MIPS architecture.

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

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.

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.

By default, most compilers for the MIPS architecture will emit additional instructions to check for the divide-by-zero and overflow cases when this instruction is used. In many compilers, the assembler mnemonic “DIV r0, rs, rt” can be used to prevent these additional test instructions to be emitted.

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

Historical Perspective:

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

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	DIV 00010	011010
SPECIAL 000000	rs	rt	rd	MOD 00011	011010
SPECIAL 000000	rs	rt	rd	DIVU 00010	011011
SPECIAL 000000	rs	rt	rd	MODU 00011	011011
SPECIAL 000000	rs	rt	rd	DDIV 00010	011110
SPECIAL 000000	rs	rt	rd	DMOD 00011	011110
SPECIAL 000000	rs	rt	rd	DDIVU 00010	011111
SPECIAL 000000	rs	rt	rd	DMODU 00011	011111
6	5	5	5	5	6

Format: `DIV MOD DIVU MODU DDIV DMOD DDIVU DMODU`

`DIV rd,rs,rt`	**MIPS32 Release 6**
`MOD rd,rs,rt`	**MIPS32 Release 6**
`DIVU rd,rs,rt`	**MIPS32 Release 6**
`MODU rd,rs,rt`	**MIPS32 Release 6**
`DDIV rd,rs,rt`	**MIPS64 Release 6**
`DMOD rd,rs,rt`	**MIPS64 Release 6**
`DDIVU rd,rs,rt`	**MIPS64 Release 6**
`DMODU rd,rs,rt`	**MIPS64 Release 6**

Purpose: Divide Integers (with result to GPR)

DIV: Divide Words Signed
MOD: Modulo Words Signed
DIVU: Divide Words Signed
MODU: Modulo Words Signed
DDIV: Divide Doublewords Signed
DMOD: Modulo Doublewords Signed
DDIVU: Divide Doublewords Signed
DMODU: Modulo Doublewords Signed

Description:

```
DIV:   GPR[rd] ← sign_extend.32( divide.signed( GPR[rs] div GPR[rt] )
MOD:   GPR[rd] ← sign_extend.32( modulo.signed( GPR[rs] mod GPR[rt] )
DIVU:  GPR[rd] ← sign_extend.32( divide.unsigned( GPR[rs] div GPR[rt] )
MODU:  GPR[rd] ← sign_extend.32( modulo.unsigned( GPR[rs] mod GPR[rt] )
DDIV:  GPR[rd] ← divide.signed( GPR[rs] div GPR[rt] )
DMOD:  GPR[rd] ← modulo.signed( GPR[rs] mod GPR[rt] )
DDIVU: GPR[rd] ← divide.unsigned( GPR[rs] div GPR[rt] )
DMODU: GPR[rd] ← modulo.unsigned( GPR[rs] mod GPR[rt] )
```

The MIPS32 Release 6 divide and modulo instructions divide the operands in GPR `rs` and GPR `rt`, and place the quotient or remainder in GPR rd.[1]

For each of the div/mod operator pairs DIV/MOD, DIVU/MODU, DDIV/DMOD, DDIVU/DMODU the results satisfy the equation `(A div B)*B + (A mod B) = A`, where `(A mod B)` has same sign as the dividend `A`, and `abs(A mod B) < abs(B)`. This equation uniquely defines the results. NOTE: if the divisor B=0, this equation cannot be satisfied, and the result is UNPREDICTABLE. This is commonly called “truncated division”.

DIV performs a signed 32-bit integer division, and places the 32-bit quotient result in the destination register.

MOD performs a signed 32-bit integer division, and places the 32-bit remainder result in the destination register. The remainder result has the same sign as the dividend.

DIVU performs an unsigned 32-bit integer division, and places the 32-bit quotient result in the destination register.

MODU performs an unsigned 32-bit integer division, and places the 32-bit remainder result in the destination register.

DDIV performs a signed 64-bit integer division, and places the 64-bit quotient result in the destination register.

DMOD performs a signed 64-bit integer division, and places the 64-bit remainder result in the destination register. The remainder result has the same sign as the dividend.

DDIVU performs an unsigned 64-bit integer division, and places the 64-bit quotient result in the destination register.

DMODU performs an unsigned 64-bit integer division, and places the 64-bit remainder result in the destination register.

Restrictions:

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

On a 64-bit CPU, the 32-bit signed divide (DIV) and modulo (MOD) instructions are UNPREDICTABLE if inputs are not signed extended 32-bit integers.

Special provision is made for the inputs to unsigned 32-bit divide and modulo on a 64-bit CPU. Since many 32-bit instructions sign extend 32 bits to 64 even for unsigned computation, properly sign extended numbers must be accepted as input, and truncated to 32 bits, clearing bits 32-63. However, it is also desirable to accept zero extended 32-bit integers, with bits 32-63 all 0.

On a 64-bit CPU, DIVU and MODU are UNPREDICTABLE if their inputs are not zero or sign extended 32-bit integers.

On a 64-bit CPU, the 32-bit divide and modulo instructions, both signed and unsigned, sign extend the result as if it is a 32-bit signed integer.

Availability:

These instructions are introduced by and required as of Release 6.

Programming Notes:

The MIPS32 Release 6 divide instructions have the same opcode mnemonic as the pre-MIPS32 Release 6 divide instructions (DIV, DIVU, DDIV, DDIVU). The instruction encodings are different, as are the instruction semantics: the MIPS32 Release 6 instruction produces only the quotient, whereas the pre-MIPS32 Release 6 instruction produces quotient and remainder in HI/LO registers respectively, and separate modulo instructions are required to obtain the remainder. The assembly syntax distinguishes the MIPS32 Release 6 from the pre-MIPS32 Release 6 divide instructions, e.g. MIPS32 Release 6 “`DIV rd,rs,rt`” specifies 3 register operands, versus pre-MIPS32 Release 6 “`DIV rs,rt`”, which has only two register arguments, with the HI/LO registers implied. Unfortunately, some

1.

assemblers accept the pseudo-instruction syntax "DIV rd,rs,rt" and expand it to do "DIV rs,rt;MFHI rd". Phrases such as "DIV with GPR output" and "DIV with HI/LO output" may be used when disambiguation is necessary.

The pre-MIPS32 Release 6 divide instructions that produce quotient and remainder in the HI/LO registers produce a Reserved Instruction Exception on MIPS32 Release 6. In the future, the instruction encoding may be reused for other instructions.

Because the MIPS divide and modulo instructions are defined to not trap if dividing by zero, it is safe to emit code that checks for zero-divide after the divide or modulo instruction. This

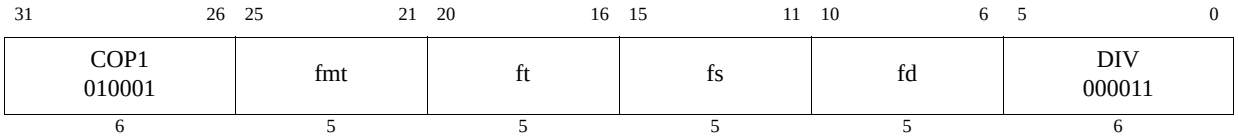
Operation

```
DIV, MOD: if NotWordValue(GPR[rs]) then UNPREDICTABLE ?
DIV, MOD: if NotWordValue(GPR[rt]) then UNPREDICTABLE ?
DIVU, MODU: if not(zero_or_sign_extended.32(GPR[rs])) then UNPREDICTABLE ?
DIVU, MODU: if not(zero_or_sign_extended.32(GPR[rt])) then UNPREDICTABLE ?

/* recommended implementation: ignore bits 32-63 for DIV, MOD, DIVU, MODU */

DIV, MOD:
  s1 ← signed_word(GPR[rs])
  s2 ← signed_word(GPR[rt])
DIVU, MODU:
  s1 ← unsigned_word(GPR[rs])
  s2 ← unsigned_word(GPR[rt])
DDIV, DMOD:
  s1 ← signed_doubleword(GPR[rs])
  s2 ← signed_dpubleword(GPR[rt])
DDIVU, DMODU:
  s1 ← unsigned_doubleword(GPR[rs])
  s2 ← unsigned_doubleword(GPR[rt])

DIV, DIVU, DDIV, DDIVU:
  quotient ← s1 div s2
MOD, MODU, DMOD, DMODU:
  remainder ← s1 mod s2

DIV:   GPR[rd] ← sign_extend.32( quotient )
MOD:   GPR[rd] ← sign_extend.32( remainder )
DIVU:  GPR[rd] ← sign_extend.32( quotient )
MODU:  GPR[rd] ← sign_extend.32( remainder )
DDIV:  GPR[rd] ← quotient
DMOD:  GPR[rd] ← remainder
DDIVU: GPR[rd] ← quotient
DMODU: GPR[rd] ← remainder
?

where

   function zero_or_sign_extended.32(val)
      if value63..32 = (value31)32 then return true
      if value63..32 = (0)32 then return true
      return false
   end function
```

Exceptions:

None[2]

2. No arithmetic exception occurs under any circumstances. Division by zero produces an UNPREDICTABLE result.



Format: `DIV.fmt`

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

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

Purpose: Floating Point Divide

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:

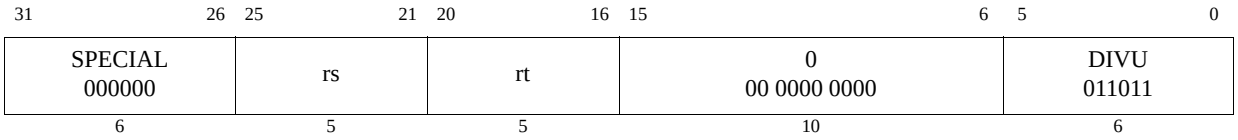
```
StoreFPR (fd, fmt, ValueFPR(fs, fmt) / ValueFPR(ft, fmt))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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



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

Purpose: Divide Unsigned Word

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**.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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:

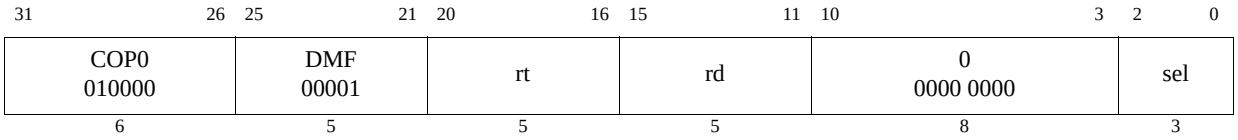
This pre-MIPS32 Release 6 instruction, DIV, which produces both quotient and remainder, has been removed[1] by MIPS32 Release 6 and has been replaced by DIV and MOD instructions that produce only quotient and remainder, respectively. Refer to the Release 6 introduced ‘DIV’ and ‘MOD’ instructions in this manual for more information. Note that this instruction remains current for all release levels lower than Release 6 of the MIPS architecture.

See “Programming Notes” for the DIV instruction.

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different 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.



Format: `DMFC0 rt, rd` **MIPS64**

`DMFC0 rt, rd, sel` **MIPS64**

Purpose: Doubleword Move from Coprocessor 0

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 **UNDEFINED** 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.

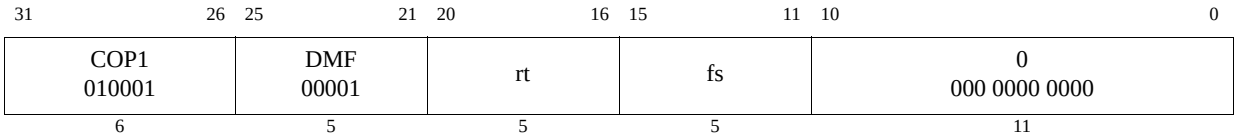
Operation:

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

Exceptions:

Coprocessor Unusable

Reserved Instruction



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

Purpose: Doubleword Move from Floating Point

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
```

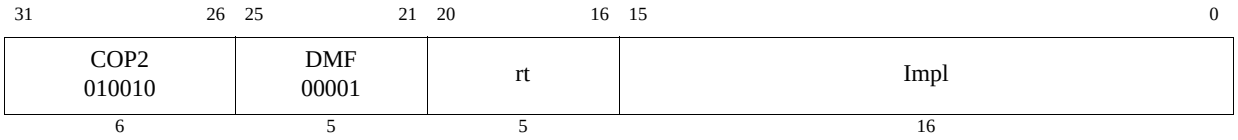
Exceptions:

Coprocessor Unusable

Reserved Instruction

Historical Information:

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



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: Doubleword Move from Coprocessor 2

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.

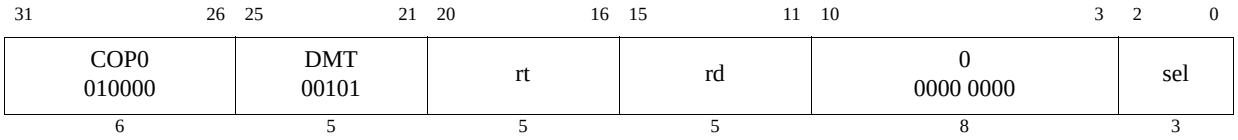
Operation:

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

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: `DMTC0 rt, rd` **MIPS64**

`DMTC0 rt, rd, sel` **MIPS64**

Purpose: Doubleword Move to Coprocessor 0

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 **UNDEFINED** 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.

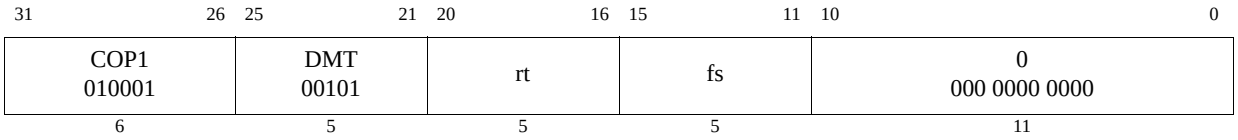
Operation:

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

Exceptions:

Coprocessor Unusable

Reserved Instruction



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

Purpose: Doubleword Move to Floating Point

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)
```

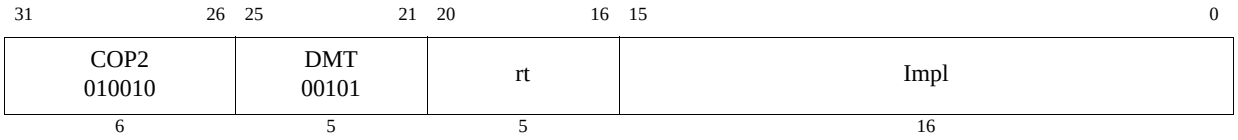
Exceptions:

Coprocessor Unusable

Reserved Instruction

Historical Information:

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



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

`DMTC2 rt, Impl, sel` **MIPS64**

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

Purpose: Doubleword Move to Coprocessor 2

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.

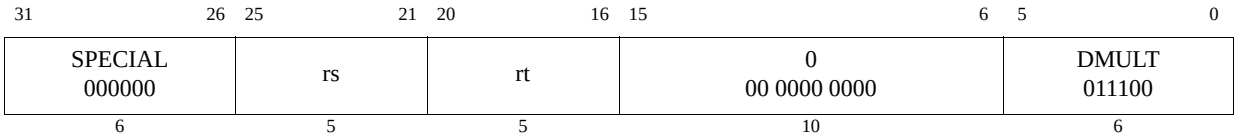
Operation:

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

Exceptions:

Coprocessor Unusable

Reserved Instruction



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

Purpose: Doubleword Multiply

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:

Availability:

This instruction has been removed in the MIPS64 Release 6 architecture.

Operation:

```
prod ← GPR[rs] × GPR[rt]
LO ← prod63..0
HI ← prod127..64
```

Exceptions:

Reserved Instruction

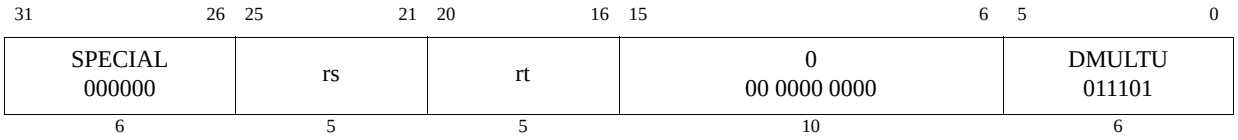
Programming Notes:

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.



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

Purpose: Doubleword Multiply Unsigned

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:

Availability:

This instruction has been removed in the MIPS64 Release 6 architecture.

Operation:

```
prod ← (0||GPR[rs]) × (0||GPR[rt])
LO ← prod63..0
HI ← prod127..64
```

Exceptions:

Reserved Instruction

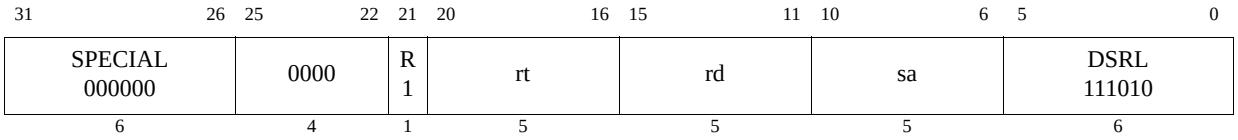
Programming Notes:

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: Doubleword Rotate Right

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*.

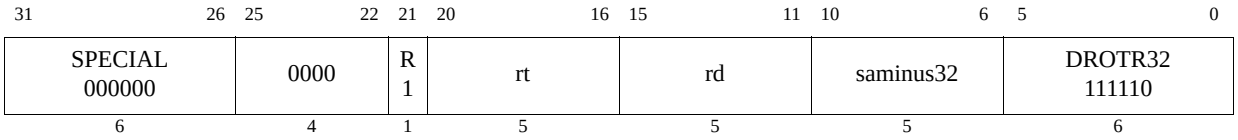
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction



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

Purpose: Doubleword Rotate Right Plus 32

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.

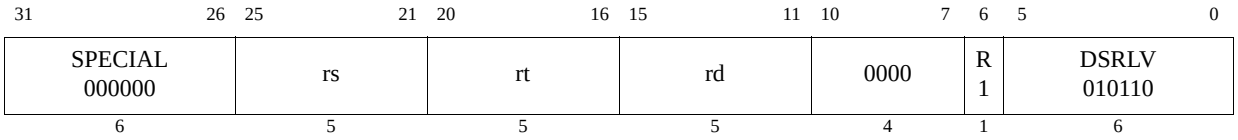
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction



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

Purpose: Doubleword Rotate Right Variable

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*.

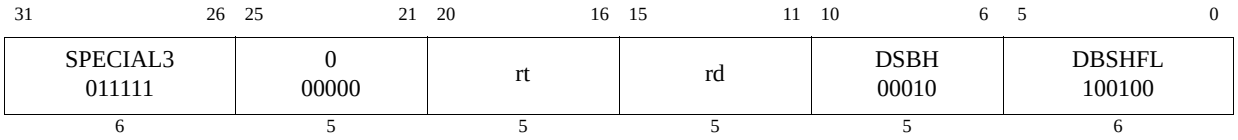
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction



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

Purpose: Doubleword Swap Bytes Within Halfwords

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:

```
GPR[rd] ← GPR[t]55..48 || GPR[t]63..56 || GPR[t]39..32 || GPR[t]47..40 ||
          GPR[t]23..16 || GPR[t]31..24 || GPR[t]7..0 || GPR[t]15..8
```

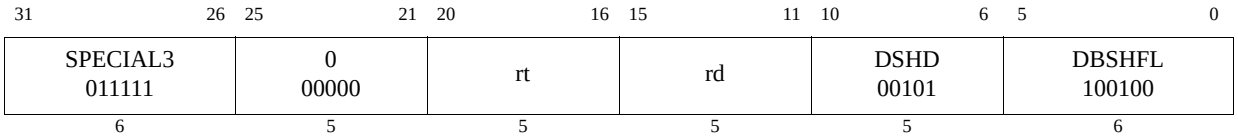
Exceptions:

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: `DSHD rd, rt` **MIPS64 Release 2**

Purpose: Doubleword Swap Halfwords Within Doublewords

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:

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

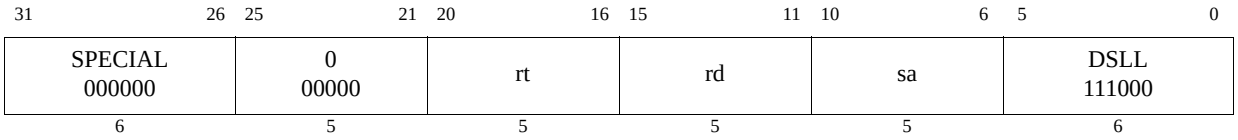
Exceptions:

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: Doubleword Shift Left Logical

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*.

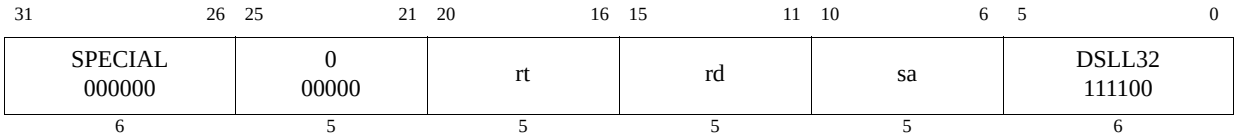
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction



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

Purpose: Doubleword Shift Left Logical Plus 32

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*.

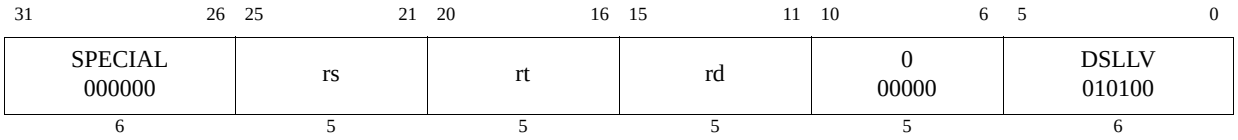
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction



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

Purpose: Doubleword Shift Left Logical Variable

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*.

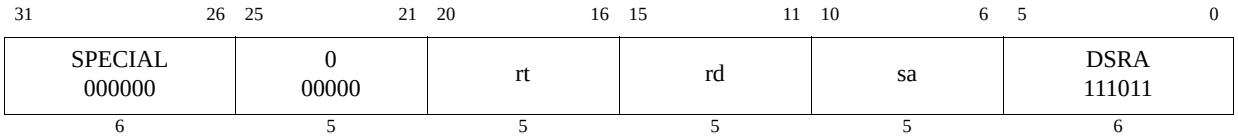
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction



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

Purpose: Doubleword Shift Right Arithmetic

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*.

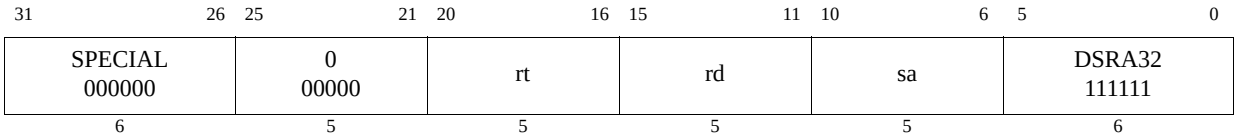
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction



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

Purpose: Doubleword Shift Right Arithmetic Plus 32

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.

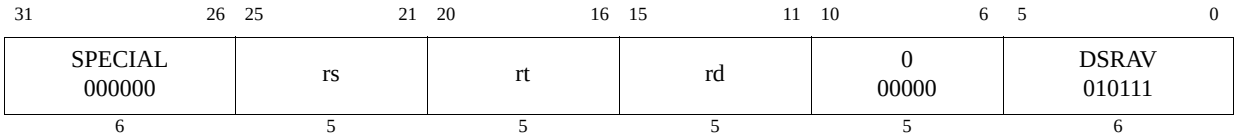
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction



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

Purpose: Doubleword Shift Right Arithmetic Variable

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*.

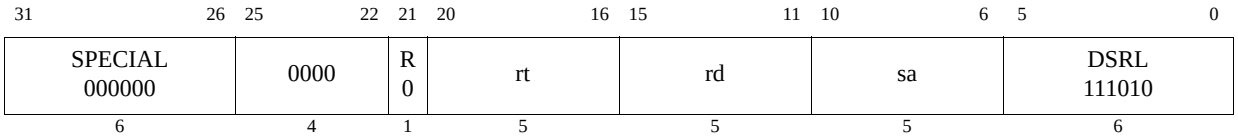
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction



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

Purpose: Doubleword Shift Right Logical

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*.

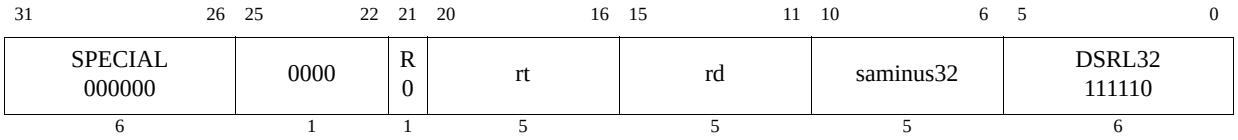
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction



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

Purpose: Doubleword Shift Right Logical Plus 32

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.

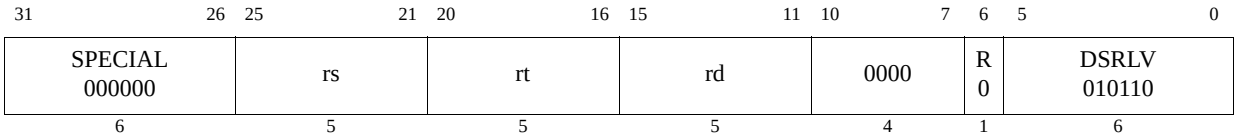
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction



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

Purpose: Doubleword Shift Right Logical Variable

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*.

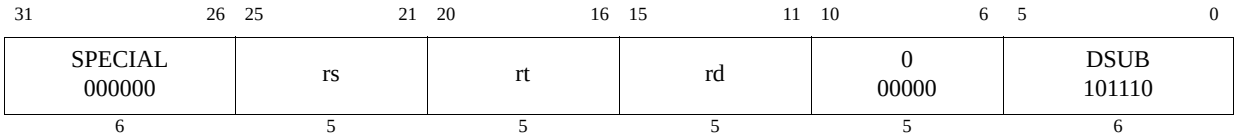
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction



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

Purpose: Doubleword Subtract

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:

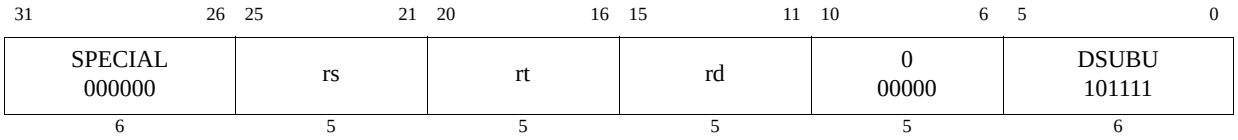
```
temp ← (GPR[rs]63||GPR[rs]) - (GPR[rt]63||GPR[rt])
if (temp64 ≠ temp63) then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp63..0
endif
```

Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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



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

Purpose: Doubleword Subtract Unsigned

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

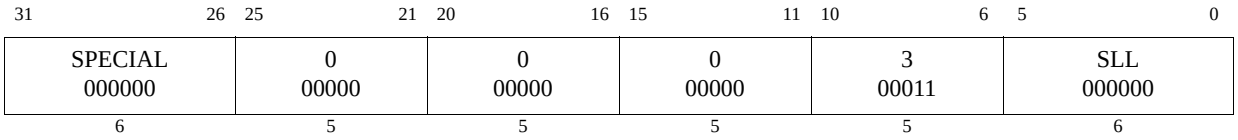
```
GPR[rd] ← GPR[rs] – GPR[rt]
```

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.



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

Purpose: Execution Hazard Barrier

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:

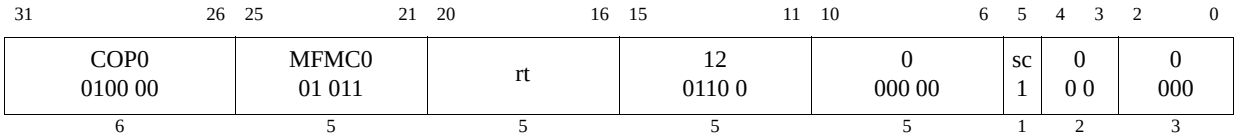
```
ClearExecutionHazards()
```

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.



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

`EI rt` **MIPS32 Release 2**

Purpose: Enable Interrupts

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
```

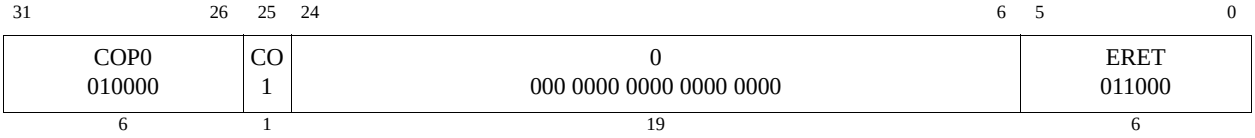
Exceptions:

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 cannot 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: Exception Return

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:

Prior to MIPS32 Release 6: The operation of the processor is **UNDEFINED** if an ERET is executed in the delay slot of a branch or jump instruction.

MIPS32 Release 6 implementations are required to signal a Reserved Instruction Exception if ERET is encountered in the delay slot or forbidden 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.

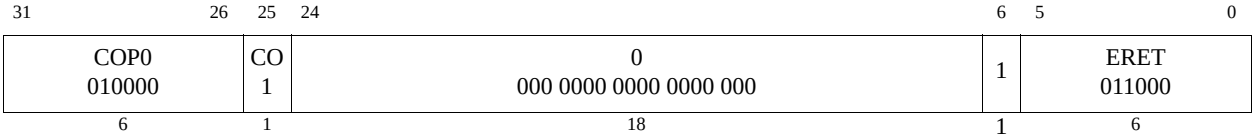
In a Release 2 implementation, ERET does not restore $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ if $Status_{BEV} = 1$, or if $Status_{ERL} = 1$ because any exception that sets $Status_{ERL}$ to 1 (Reset, Soft Reset, NMI, or cache error) does not save $SRSCtl_{CSS}$ in $SRSCtl_{PSS}$. If software sets $Status_{ERL}$ 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() | (Config3_ISA > 0) then
   PC ← temp_63..1 || 0
   ISAMode ← temp_0
else
   PC ← temp
endif
LLbit ← 0
ClearHazards()
```

Exceptions:

Coprocessor Unusable Exception



Format: `ERETNC` **MIPS32 Release 5**

Purpose: Exception Return No Clear

To return from interrupt, exception, or error trap without clearing the LLbit.

Description:

ERETNC clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ when implemented, and returns to the interrupted instruction at the completion of interrupt, exception, or error processing. ERETNC does not execute the next instruction (i.e., it has no delay slot).

ERETNC is identical to ERET except that an ERETNC will not clear the LLbit that is set by execution of an LL instruction, and thus when placed between an LL and SC sequence, will never cause the SC to fail.

An ERET should continue to be used by default in interrupt and exception processing handlers: the handler may have accessed a synchronizable block of memory common to code that is atomically accessing the memory, and where the code caused the exception or was interrupted. Similarly, a process context-swap must also continue to use an ERET in order to avoid a possible false success on execution of SC in the restored context.

Multiprocessor systems with non-coherent cores (i.e., without hardware coherence snooping) should also continue to use ERET, since it is the responsibility of software to maintain data coherence in the system.

An ERETNC is useful in cases where interrupt/exception handlers and kernel code involved in a process context-swap can guarantee no interference in accessing synchronizable memory across different contexts. ERETNC can also be used in an OS-level debugger to single-step through code for debug purposes, avoiding the false clearing of the LLbit and thus failure of an LL and SC sequence in single-stepped code.

Software can detect the presence of ERETNC by reading $Config5_{LLB}$.

Restrictions:

MIPS32 Release 6 implementations are required to signal a Reserved Instruction Exception if ERET is encountered in the delay slot or forbidden slot of a branch or jump instruction.

ERETNC 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 in the PC to which the ERETNC returns.

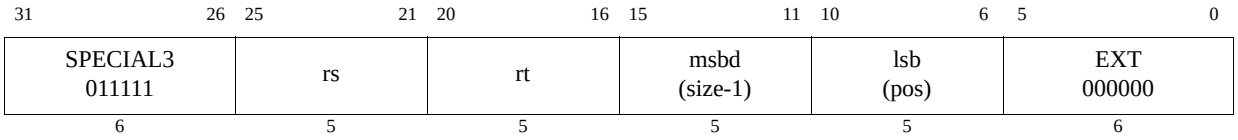
Operation:

```
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() | (Config3_ISA > 0) then
```

```
        PC ← temp63..1 || 0
        ISAMode ← temp0
    else
        PC ← temp
    endif
    ClearHazards()
```

Exceptions:

Coprocessor Unusable Exception



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

Purpose: Extract Bit Field

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:

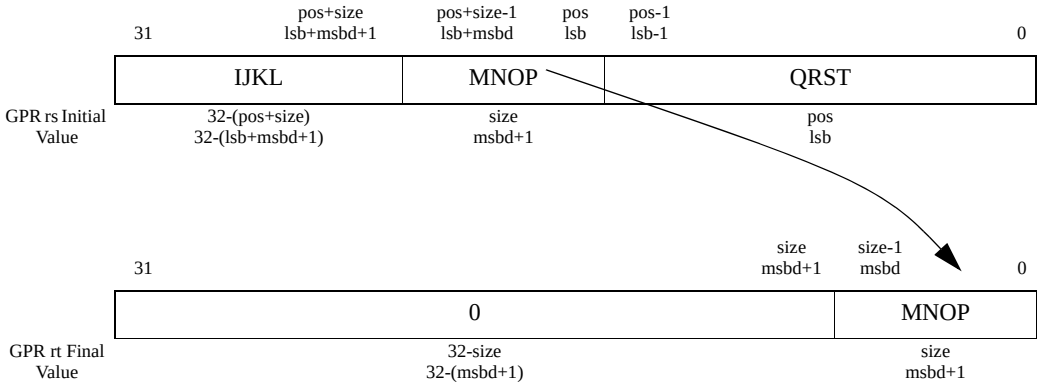
```
msbd ← size-1
lsb ← pos
```

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 3.11 Operation of the EXT 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+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:

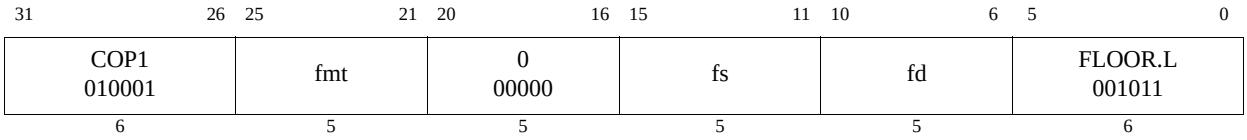
```
if ((lsb + msbd) > 31) or (NotWordValue(GPR[rs])) then
```

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

Exceptions:

Reserved Instruction

33 31



Format: `FLOOR.L.fmt`

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

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

Purpose: Floating Point Floor Convert to Long Fixed Point

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

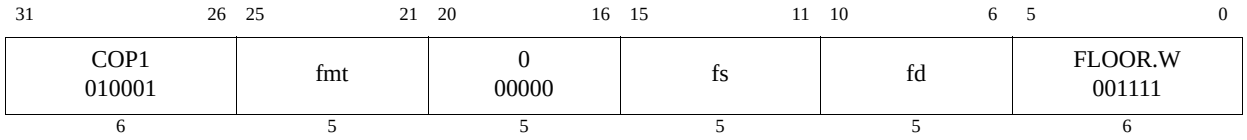
```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact



Format: `FLOOR.W.fmt`

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

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

Purpose: Floating Point Floor Convert to Word Fixed Point

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:

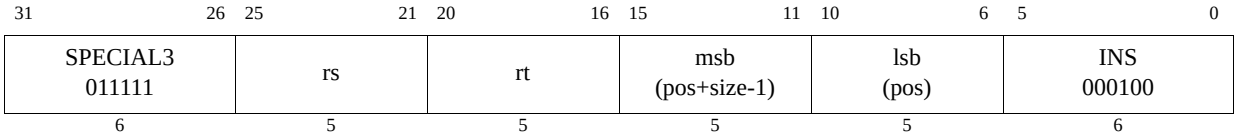
```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact



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

Purpose: Insert Bit Field

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:

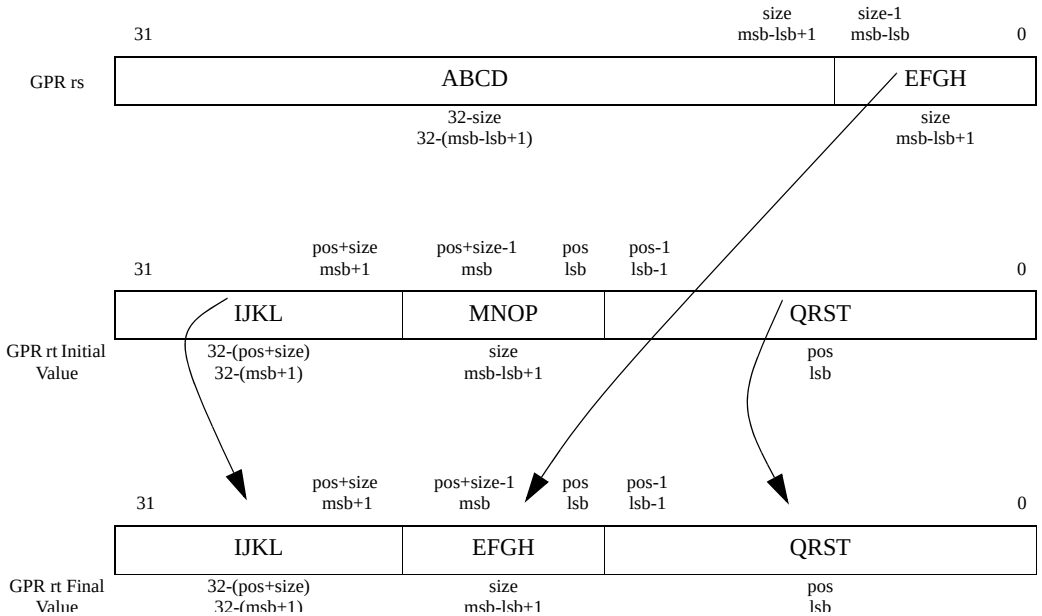
```
msb ← pos+size-1
lsb ← pos
```

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 3.12 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*.

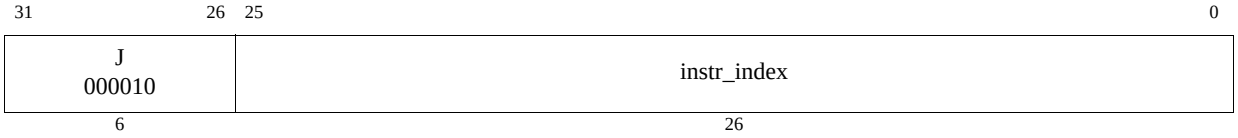
If either GPR *rs* or GPR *rt* does not contain sign-extended 32-bit values (bits **63..31** equal), then the result of the operation is **UNPREDICTABLE**.

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



Format: `J target` **MIPS32**

Purpose: Jump

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 2bits. 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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Operation:

```
I:
I+1: PC ← PC_GPRLEN-1..28 || instr_index || 0^2
```

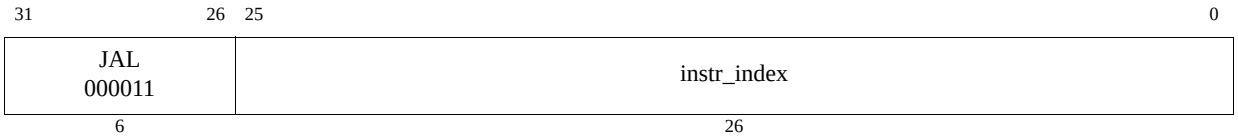
Exceptions:

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 256MB region aligned on a 256MB 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 256MB region, it can branch only to the following 256MB region containing the branch delay slot.



Format: `JAL target` **MIPS32**

Purpose: Jump and Link

To execute a procedure call within the current 256MB-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” 256MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2bits. 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:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Operation:

```
I:    GPR[31] ← PC + 8
I+1:  PC ← PC_GPRLEN-1..28 || instr_index || 0^2
```

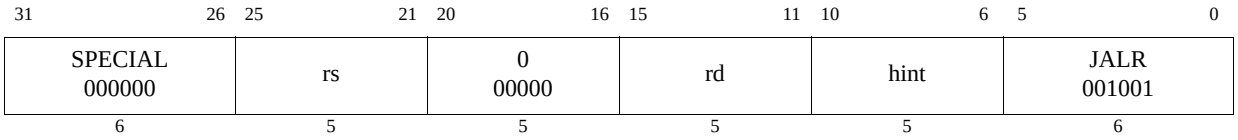
Exceptions:

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 256MB region aligned on a 256MB 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 256MB region, it can branch only to the following 256MB region containing the branch delay slot.



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

`JALR rd, rs` **MIPS32**

Purpose: Jump and Link Register

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 nor microMIPS32/64 ISA:

- 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. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

For processors that do implement the MIPS16e ASE or microMIPS32/64 ISA:

- 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.

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR *rs*.

For processors that do not implement the microMIPS32/64 ISA, the effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16e ASE nor microMIPS32/64 ISA, 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 or microMIPS32/64 ISA, if target ISAMode bit is 0 (GPR *rs* bit 0) is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Operation:

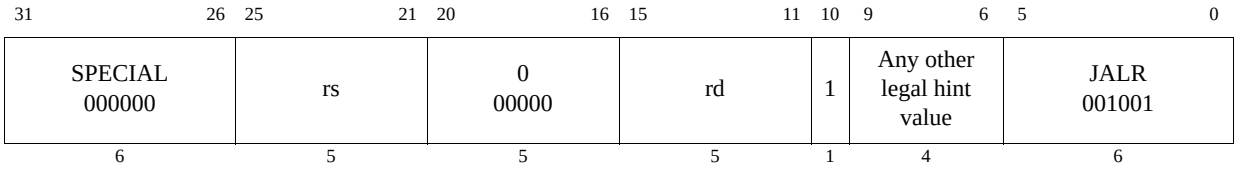
```
I:  temp ← GPR[rs]
    GPR[rd] ← PC + 8
I+1:if Config1CA = 0 then
        PC ← temp
    else
        PC ← tempGPRLEN-1..1 || 0
        ISAMode ← temp0
    endif
```

Exceptions:

None

Programming Notes:

This branch-and-link instruction that can select a register for the return link; 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: Jump and Link Register with Hazard Barrier

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 nor microMIPS32/64 ISA:

- 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. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

For processors that do implement the MIPS16 ASE or microMIPS32/64 ISA:

- 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.

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR *rs*.

For processors that do not implement the microMIPS32/64 ISA, the effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16 ASE nor microMIPS32/64 ISA, 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 or microMIPS32/64 ISA, 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 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.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

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:

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

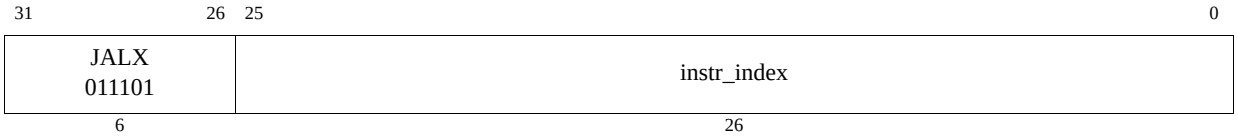
In particular, as of MIPS32 Release 6 `JR.HB rs` is implemented as `JALR.HB r0,rs`, i.e. as `JALR.HB` with the destination set to the zero register, `r0`.

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: `JALX target` **MIPS64 with (microMIPS64 or MIPS16e)**

Purpose: Jump and Link Exchange

To execute a procedure call within the current 256 MB-aligned region and change the *ISA Mode* from MIPS64 to microMIPS64 or MIPS16e.

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. The value stored in GPR 31 bit 0 reflects the current value of the *ISA Mode* bit.

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

Restrictions:

This instruction only supports 32-bit aligned branch target addresses.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

If the microMIPS base architecture is not implemented and the MIPS16e ASE is not implemented, a Reserved Instruction Exception is initiated.

MIPS32 Release 6 removes the JALX instruction, reusing its opcode immediately for DAUI. I.e. pre-MIPS32 Release 6 code using JALX cannot run on MIPS32 Release 6 by trap-and-emulate.

Operation:

```
I:      GPR[31] ← PC + 8
I+1:    PC ← PC_GPRLEN-1..28 || instr_index || 0^2
        ISAMode ← (not ISAMode)
```

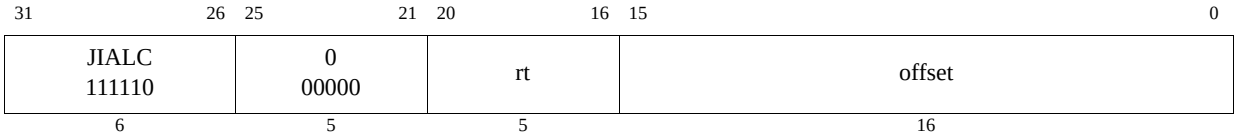
Exceptions:

None

Programming Notes:

Forming the branch target address by concatenating 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: `JIALC`
`JIALC target`

MIPS32 Release 6

Purpose: Jump Indexed and Link, Compact

Description: `r31 ← PC+4, PC ←memory_address( rt + sign_extend( offset ) )`

The branch target is formed by sign extending the offset field of the instruction and adding it to the contents of GPR `rt`.

The offset is NOT shifted, i.e. each bit of the offset is added to the corresponding bit of the GPR.

Places the return address link in GPR 31. The return link is the address of the following instruction, where execution continues after a procedure call returns.

Restrictions:

This instruction is an unconditional, always taken, compact jump, and hence has neither a Delay Slot nor a Forbidden Slot. The instruction after the jump is NOT executed.

Availability:

This instruction is introduced by and required as of MIPS32 Release 6.

Exceptions:

None[1]

Operation:

```
GPR[31] ← PC + 4
PC ← memory_address( GPR[rt] + sign_extend(target_offset) )
```

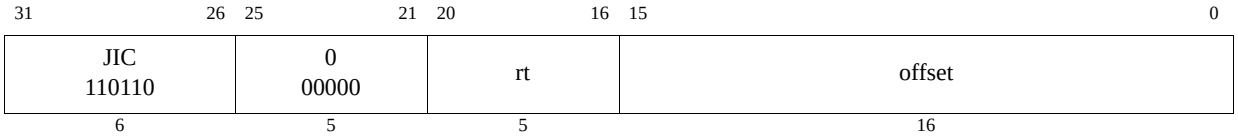
Special Considerations:

Note the special restrictions on relative addresses that cross the 2G / 32-bit signed boundaries, formulated as pseudo-code function `memory_address`.

Programming Notes:

JIALC does NOT shift the offset before adding it the register. This can be used to eliminate tags in the least significant bits that would otherwise produce misalignment.

1. Except for possible Reserved Instruction Exception if used in a Forbidden Slot, true of these and most other control transfer instructions (CTIs)



Format: `JIC`
`JIC offset16`

MIPS32 Release 6

Purpose: Jump Indexed, Compact

Description: `PC ← memory_address( rt + sign_extend( offset ) )`

The branch target is formed by sign extending the offset field of the instruction and adding it to the contents of GPR `rt`.

The offset is NOT shifted, i.e. each bit of the offset is added to the corresponding of the GPR.

Restrictions:

This instruction is an unconditional, always taken, compact jump, and hence has neither a Delay Slot nor a Forbidden Slot. The instruction after the jump is NOT executed.

Availability:

This instruction is introduced by and required as of MIPS32 Release 6.

Exceptions:

None[1]

Operation:

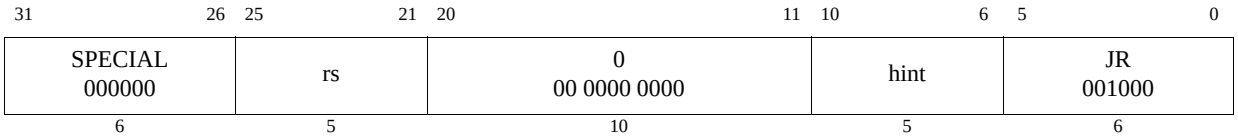
```
PC ← memory_address( GPR[rt] + sign_extend(target_offset) )
```

Special Considerations:

Note the special restrictions on relative addresses that cross the 2G / 32-bit signed boundaries, formulated as pseudo-code function `memory_address`.

JIC does NOT shift the offset before adding it the register. This can be used to eliminate tags in the least significant bits that would otherwise produce misalignment.

1. Except for possible Reserved Instruction Exception if used in a Forbidden Slot, true of these and most other control transfer instructions (CTIs)



Format: `JR rs` **MIPS32**

Purpose: Jump Register

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 or microMIPS32/64 ISA, 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:

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR *rs*.

For processors that do not implement the microMIPS ISA, the effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16e ASE or microMIPS ISA, 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 or microMIPS ISA, 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.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Operation:

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

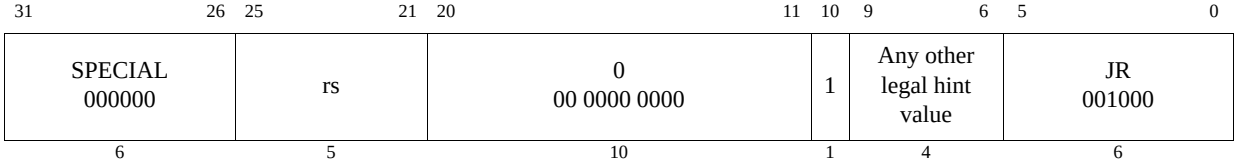
Exceptions:

None

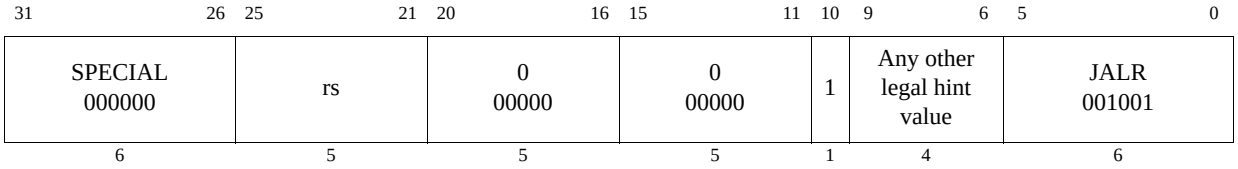
Programming Notes:

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.

MIPS32, (all release levels less than Release 6)



MIPS32 Release 6 (Release 6 and future)



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

Purpose: Jump Register with Hazard Barrier

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 MIPS16e ASE or microMIPS32/64 ISA, 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:

Pre-MIPS32 Release 6 JR.HB has a distinct function code. As of MIPS32 Release 6, `JR.HB rs` is implemented as `JALR.HB r0,rs`, i.e. as `JALR.HB` with the destination set to the zero register, `r0`. The pre-MIPS32 Release 6 instruction encoding for `JR.HB` has been removed in MIPS32 Release 6. Assemblers should accept the JR.HB syntax.

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR *rs*.

For processors that do not implement the microMIPS ISA, the effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16 ASE or microMIPS ISA, if either of the two least-signifi-

cant 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 or microMIPS ISA, 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 JR.HB. Only hazards created by instructions executed before the JR.HB are cleared by the JR.HB.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

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 */
```

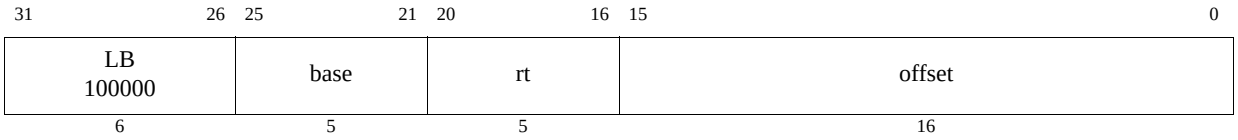
```
    or     v0, v0, a0          /* OR in new ASID value */
    mtc0   v0, C0_EntryHi      /* Rewrite EntryHi with new ASID */
    jr.hb  ra                  /* Return, clearing the hazard */
    nop
```

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:
```



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

Purpose: Load Byte

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:

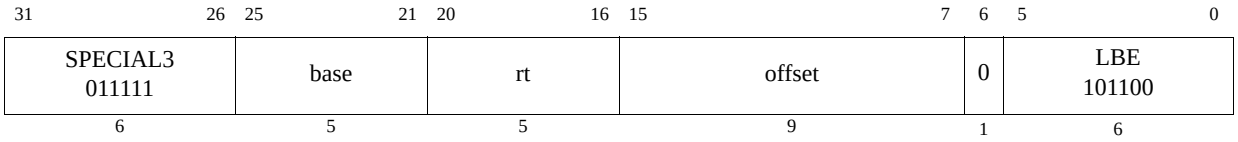
None

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



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

Purpose: Load Byte EVA

To load a byte as a signed value from user mode virtual address space when executing in kernel mode.

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

The LBE instruction functions in exactly the same fashion as the LB instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode and executing in kernel mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Restrictions:

Only usable when access to Coprocessor0 is enabled and accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

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

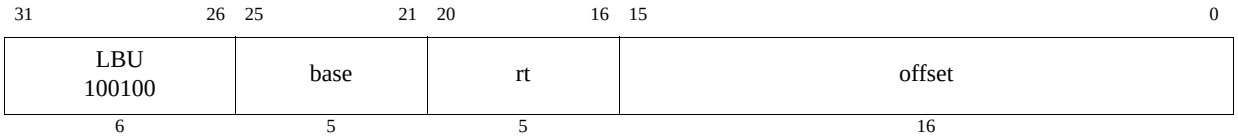
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



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

Purpose: Load Byte Unsigned

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:

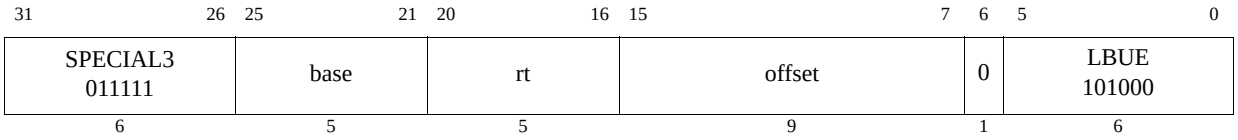
None

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



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

Purpose: Load Byte Unsigned EVA

To load a byte as an unsigned value from user mode virtual address space when executing in kernel mode.

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

The LBUE instruction functions in exactly the same fashion as the LBU instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Restrictions:

Only usable when access to Coprocessor0 is enabled and accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

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

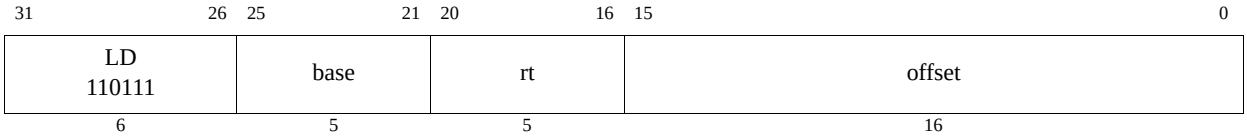
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



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

Purpose: Load Doubleword

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:

Prior to MIPS32 Release 6: 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.

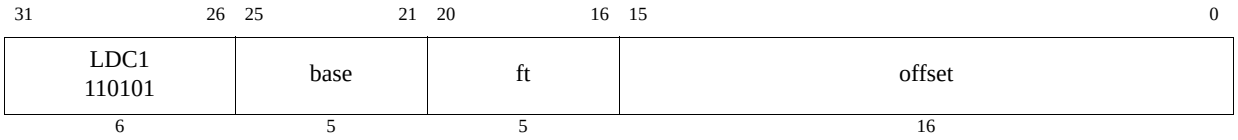
MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
       if not MisalignedSupport()
          then SignalException(AddressError)
          else /* implementation dependent misalignment handling */
       endif
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



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

Purpose: Load Doubleword to Floating Point

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:

Prior to MIPS32 Release 6: An Address Error exception occurs if $EffectiveAddress_{2..0} \neq 0$ (not doubleword-aligned).

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

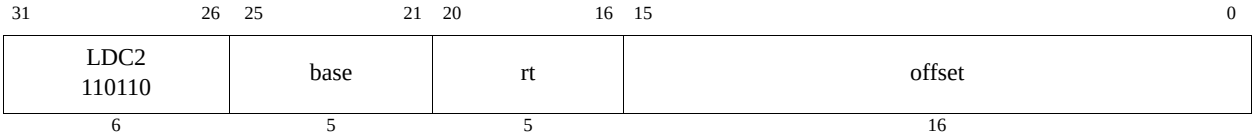
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
          if not MisalignedSupport()
              then SignalException(AddressError)
              else /* implementation dependent misalignment handling */
          endif
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
                                    StoreFPR(ft, UNINTERPRETED_DOUBLEWORD,
memdoubleword)
```

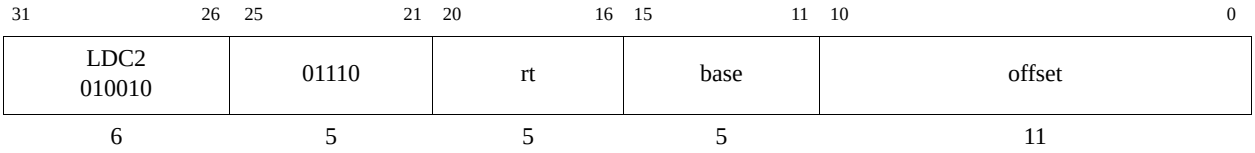
Exceptions:

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

MIPS32, (all release levels less than Release 6)



MIPS Release 6 (Release 6 and future)



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

Purpose: Load Doubleword to Coprocessor 2

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 signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

Prior to MIPS Release 6: An Address Error exception occurs if EffectiveAddress$_{2..0} \neq 0$ (not doubleword-aligned).

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

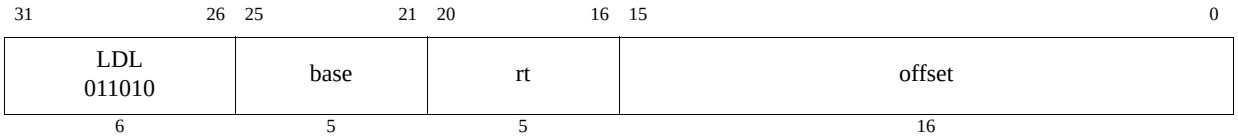
```
vAddr ← sign_extend(offset) + GPR[base]
   if vAddr2..0 ≠ 03 then
          if not MisalignedSupport()
              then SignalException(AddressError)
              else /* implementation dependent misalignment handling */
          endif
       endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
CPR[2,rt,0] ← memdoubleword
```

`CPR[2,rt,0]` `CPR[2,rt+1,0]` **Exceptions:**

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

Programming Notes:

As shown in the instruction drawing above, the MIPS Release 6 architecture implements an 11-bit offset, whereas all release levels lower than Release 6 of the MIPS architecture implement a 16-bit offset.



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

Purpose: Load Doubleword Left

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 3.13 Unaligned Doubleword Load Using LDL and LDR

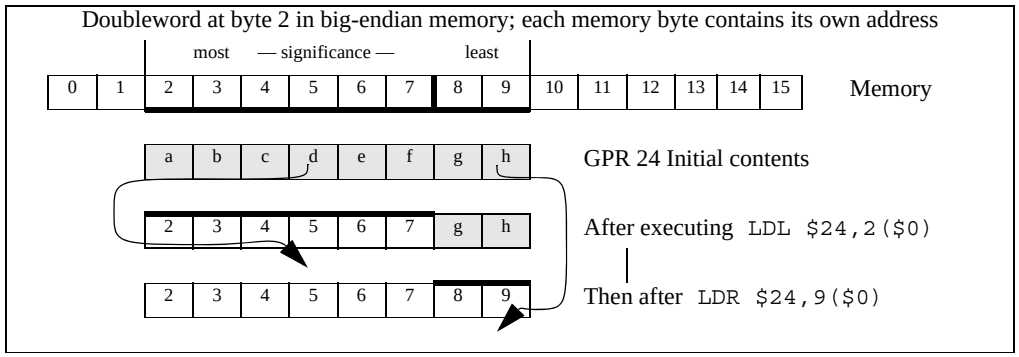
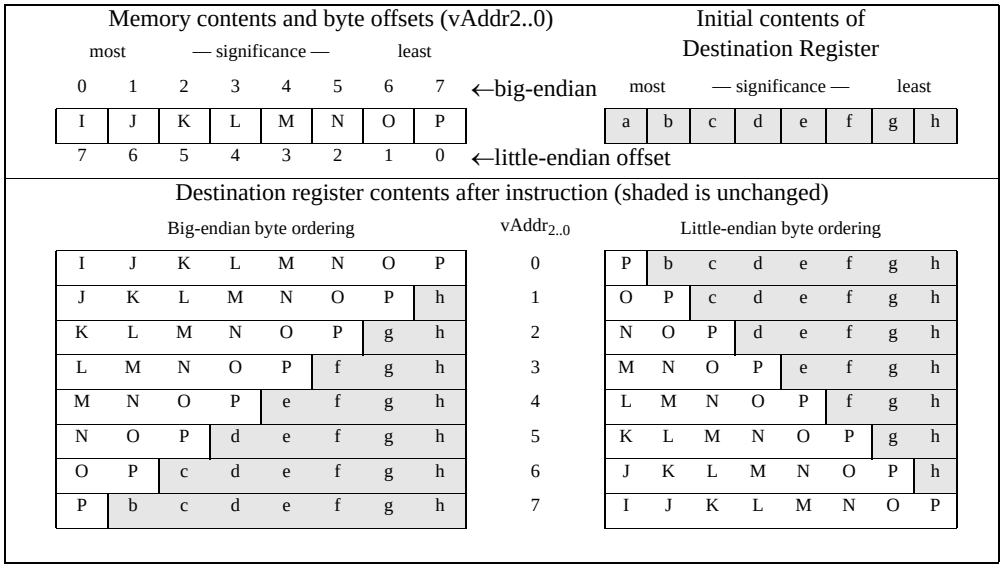


Figure 3-11 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 6 bytes, is located in the aligned doubleword starting with the most-significant byte at 2. LDL first loads these 6 bytes into the 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 3.14 Bytes Loaded by LDL Instruction



Restrictions:

Availability:

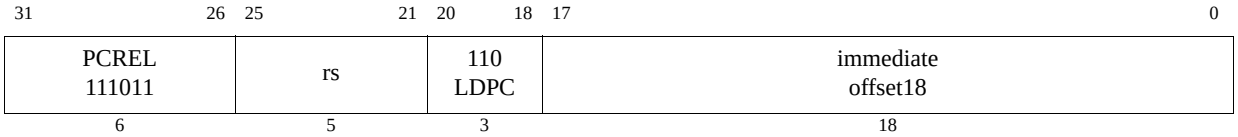
Release 6 removes the load/store-left/right family of instructions, and requires the system to support misaligned memory accesses.

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



Format: `LDPC`
`LDPC rs, offset18` **MIPS64 Release 6**

Purpose: Load Doubleword PC-relative

To load a doubleword from memory, using a PC-relative address.

Description: `GPR[rs] ← memory[ PC + sign_extend( offset18 << 3) ]`

The 18 bit offset is shifted left by 3 bits, sign-extended, and added to the address of the LDPC instruction.

The contents of the 64-bit doubleword at the memory location specified by the effective address are fetched, and placed in GPR *rt*.

LDPC, like all memory references, applies `memory_address` to the virtual address, sign extending if User mode and Status.UX=0. **Restrictions:**

LDPC is naturally aligned, by specification.

Availability:

LDPC is required as of MIPS64 Release 6.

Operation

```
vAddr ← memory_address( sign_extend(offset) + GPR[base] )
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[rt] ← memdoubleword
```

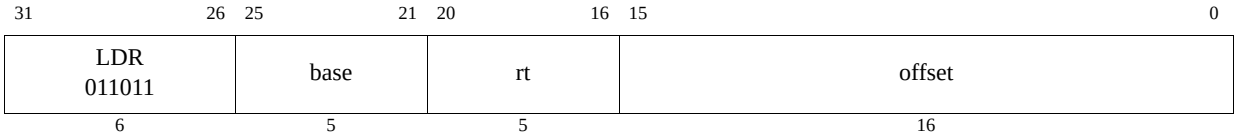
Exceptions:

TLB Refill, TLB Invalid, TLB Read Inhibit, Bus Error, Watch, Reserved Instruction

Programming Note

The MIPS32 Release 6 PC-relative loads (LWPC, LWUPC, LDPC) are considered data references.

For the purposes of watchpoints (provided by the CP0 *WatchHi* and *WatchLo* registers) and EJTAG breakpoints, the PC-relative reference is considered to be a data reference, rather than an instruction reference. That is, the watchpoint or breakpoint is triggered only if enabled for data references.



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

Purpose: Load Doubleword Right

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

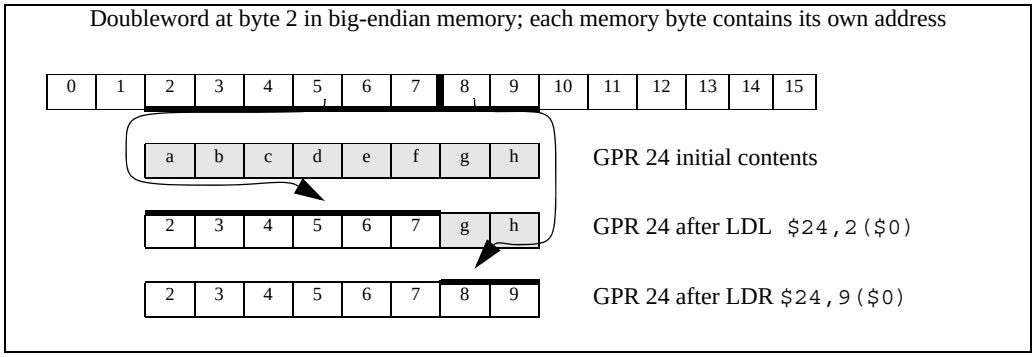
Description: `GPR[rt] ← GPR[rt] MERGE memory[GPR[base] + offset]`

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.

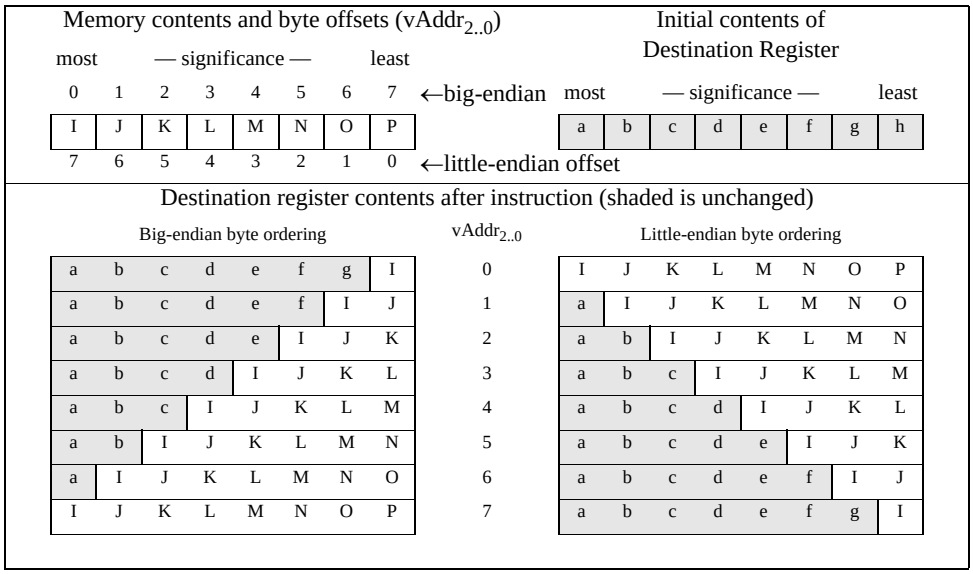
Figure 3.15 Unaligned Doubleword Load Using LDR and LDL



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 3.16 Bytes Loaded by LDR Instruction



Restrictions:

Availability:

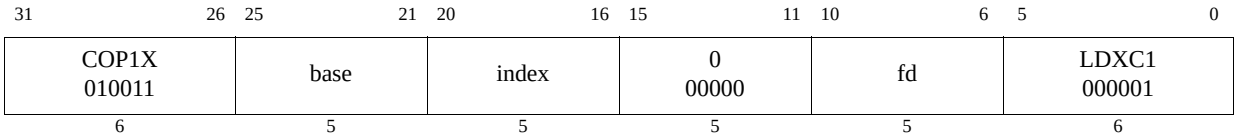
Release 6 removes the load/store-left/right family of instructions, and requires the system to support misaligned memory accesses.

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



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

Purpose: Load Doubleword Indexed to Floating Point

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:

This instruction has been removed[1] in the MIPS32 Release 6 architecture.

Prior to MIPS32 Release 6, the following restrictions apply:

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

Availability:

Removed in MIPS32 Release 6.

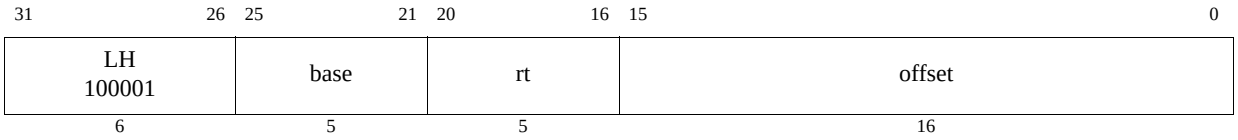
Operation:

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

Exceptions:

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

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Load Halfword

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:

Prior to MIPS32 Release 6: The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

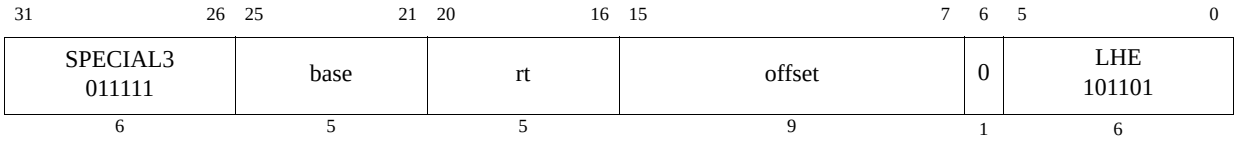
MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   if not MisalignedSupport()
      then SignalException(AddressError)
      else /* implementation dependent misalignment handling */
   endif
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



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

Purpose: Load Halfword EVA

To load a halfword as a signed value from user mode virtual address space when executing in kernel mode.

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

The LHE instruction functions in exactly the same fashion as the LH instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Restrictions:

Only usable when access to Coprocessor0 is enabled and accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Prior to MIPS32 Release 6: The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   if not MisalignedSupport()
      then SignalException(AddressError)
      else /* implementation dependent misalignment handling */
   endif
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian^2 || 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

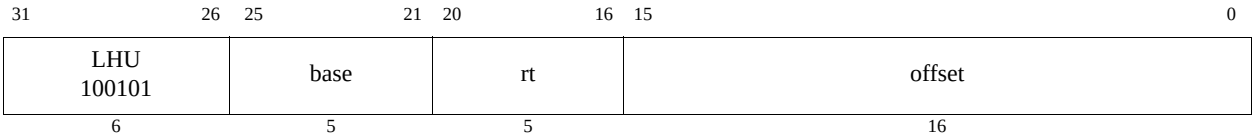
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



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

Purpose: Load Halfword Unsigned

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:

Prior to MIPS32 Release 6: The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

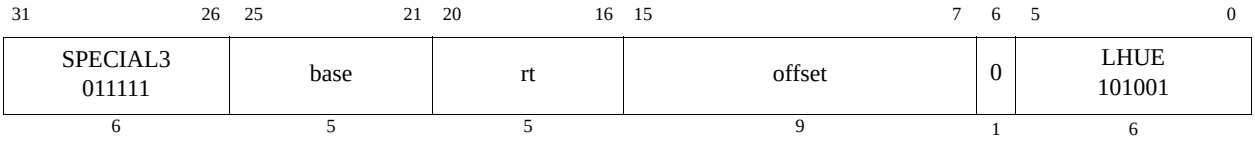
MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   if not MisalignedSupport()
      then SignalException(AddressError)
      else /* implementation dependent misalignment handling */
   endif
endif

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: `LHUE rt, offset(base)` **MIPS32**

Purpose: Load Halfword Unsigned EVA

To load a halfword as an unsigned value from user mode virtual address space when executing in kernel mode.

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

The LHUE instruction functions in exactly the same fashion as the LHU instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Restrictions:

Only usable when access to Coprocessor0 is enabled and accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Pre-MIPS32 Release 6: The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   if not MisalignedSupport()
      then SignalException(AddressError)
      else /* implementation dependent misalignment handling */
   endif
endif

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian^2 || 0))
memdoubleword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr2..0 xor (BigEndianCPU^2 || 0)
GPR[rt] ← zero_extend(memdoubleword15+8*byte..8*byte)
```

Exceptions:

TLB Refill

TLB Invalid

Bus Error

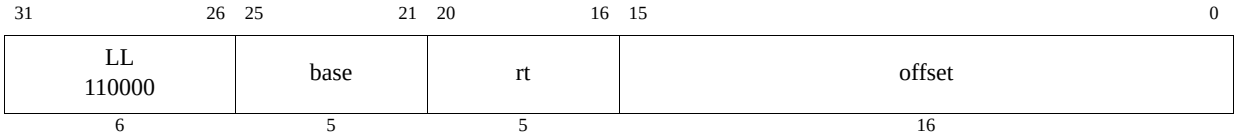
Address Error

Watch

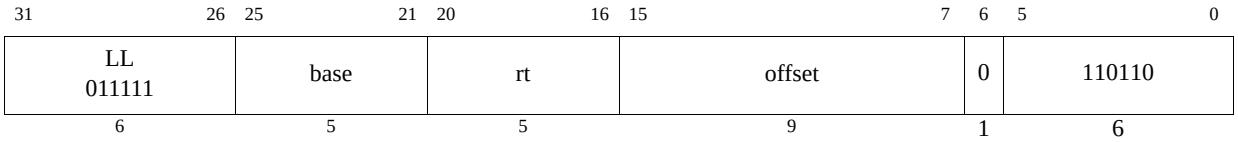
Reserved Instruction

Coprocessor Unusable

MIPS32, (all release levels less than Release 6)



MIPS32 Release 6 (Release 6 and future)



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

Purpose: Load Linked Word

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 is **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.

MIPS Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (AddressError) on a misaligned access is required.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
    SignalException(AddressError)
```

```
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor (ReverseEndian || 0^2))
memdoubleword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte ← vAddr_2..0 xor (BigEndianCPU || 0^2)
GPR[rt] ← sign_extend(memdoubleword_31+8*byte..8*byte)
LLbit ← 1
```

Exceptions:

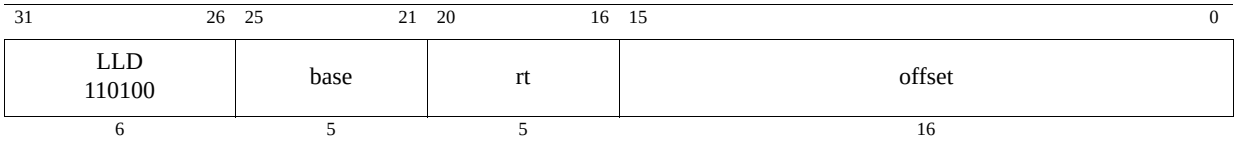
TLB Refill, TLB Invalid, Address Error, Watch

Programming Notes:

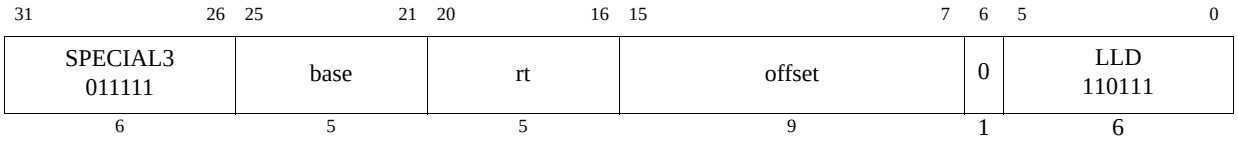
There is no Load Linked Word Unsigned operation corresponding to Load Word Unsigned.

As shown in the instruction drawing above, the MIPS Release 6 architecture implements a 9-bit offset, whereas all release levels lower than Release 6 of the MIPS architecture implement a 16-bit offset.

MIPS64, (all release levels less than Release 6)



MIPS64 (Release 6 and future)



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

Purpose: Load Linked Doubleword

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 nonzero, an Address Error exception occurs.

MIPS64 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (AddressError) on a misaligned access is required.

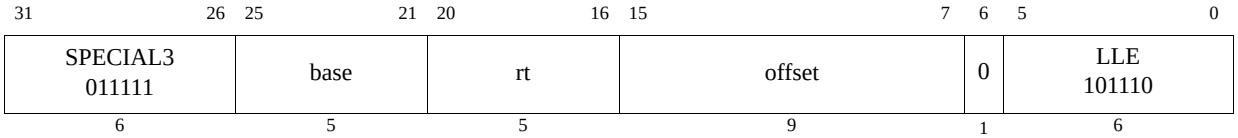
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
   SignalException(AddressError)
endif
```

```
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[rt] ← memdoubleword
LLbit ← 1
```

Exceptions:

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



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

Purpose: Load Linked Word EVA

To load a word from a user mode virtual address when executing in kernel mode for an atomic read-modify-write

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

The LLE and SCE instructions provide the primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations using user mode virtual addresses while executing in kernel mode.

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 LLE is executed it starts an active RMW sequence replacing any other sequence that was active. The RMW sequence is completed by a subsequent SCE instruction that either completes the RMW sequence atomically and succeeds, or does not and fails.

Executing LLE on one processor does not cause an action that, by itself, causes an SCE for the same block to fail on another processor.

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

The LLE instruction functions in exactly the same fashion as the LL instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Segmentation Control for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Restrictions:

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

Prior to MIPS32 Release 6: 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.

MIPS Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (AddressError) on a misaligned access is required.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
```

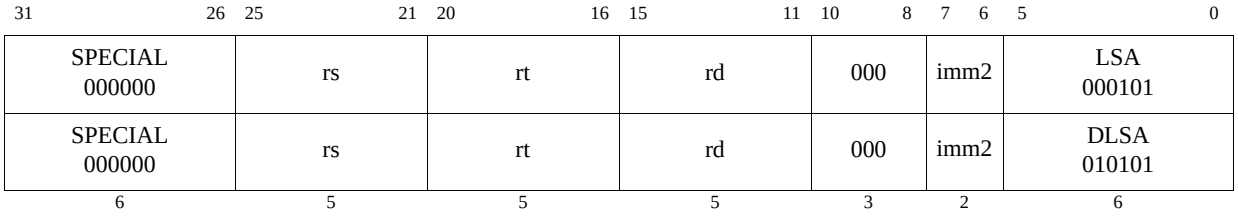
```
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor (ReverseEndian || 0^2))
memdoubleword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte ← vAddr_2..0 xor (BigEndianCPU || 0^2)
GPR[rt] ← sign_extend(memdoubleword_31+8*byte..8*byte)
LLbit ← 1
```

Exceptions:

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

Programming Notes:

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



31 26	25 21	20 16	15 11	10 8	7 6	5 0
SPECIAL 000000	rs	rt	rd	000	imm2	LSA 000101
SPECIAL 000000	rs	rt	rd	000	imm2	DLSA 010101
6	5	5	5	3	2	6

Format: `LSA DLSA`
`LSA rd,rs,rt,imm2` **MIPS32 Release 6**
`DLSA rd,rs,rt,imm2` **MIPS64 Release 6**

Purpose: Load Scaled Address, Doubleword Load Scaled Address

Description:

```
LSA:   GPR[rd] ← sign_extend.32( GPR[rs] << (imm2+1) + GPR[rt] )
DLSA:  GPR[rd] ← GPR[rs] << (imm2+1) + GPR[rt]
```

LSA adds two values derived from registers `rs` and `rt`, with an optional scaling shift on `rs` formed by adding 1 to the `imm2` field, which is interpreted as unsigned. I.e. the scaling left shift varies from 1 to 5, corresponding to multiplicative scaling values of ×2, ×4, ×8, ×16, bytes, or 16, 32, 64, or 128 bits.

LSA is a MIPS32 compatible instruction, sign extending its result from bit 31 to bit 63.

DLSA is a MIPS64 compatible instruction, performing the scaled index calculation fully 64-bits wide.

Restrictions:

Availability:

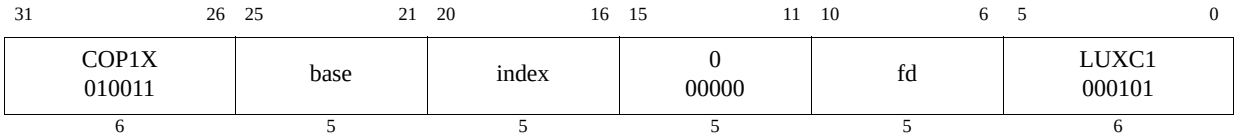
LSA and DLSA are introduced by and required as of Release 6.

Operation

```
LSA:   GPR[rd] ← sign_extend.32( GPR[rs] << (imm2+1) + GPR[rt] )
DLSA:  GPR[rd] ← GPR[rs] << (imm2+1) + GPR[rt]
```

Exceptions:

None.



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

Purpose: Load Doubleword Indexed Unaligned to Floating Point

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:

This instruction has been removed[1] in the MIPS32 Release 6 architecture.

Prior to MIPS32 Release 6, the following restrictions apply:

The result of this instruction is **UNPREDICTABLE** if the processor is executing in the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

This instruction has been removed in the Release 6 architecture.

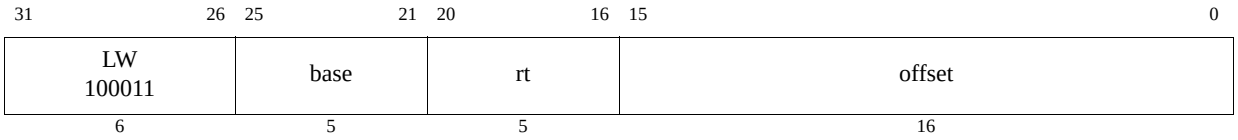
Operation:

```
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

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Load Word

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:

Prior to MIPS32 Release 6: 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.

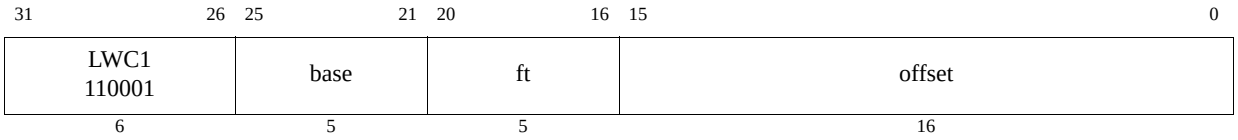
MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
   if not MisalignedSupport()
      then SignalException(AddressError)
      else /* implementation dependent misalignment handling */
   endif
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] ← sign_extend(memdoubleword31+8*byte..8*byte)
```

Exceptions:

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



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

Purpose: Load Word to Floating Point

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*. If FPRs are 64 bits wide, bits 63..32 of FPR *ft* become **UNPREDICTABLE**. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

Prior to MIPS32 Release 6: An Address Error exception occurs if $EffectiveAddress_{1..0} \neq 0$ (not word-aligned).

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

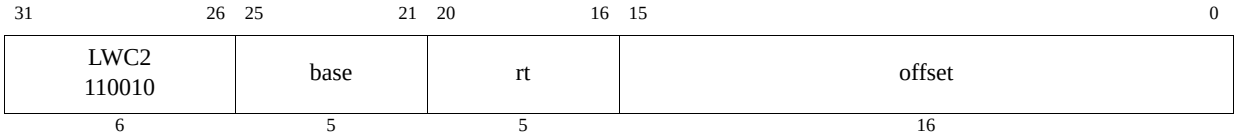
Operation:

```
        vAddr ← sign_extend(offset) + GPR[base]
        if vAddr1..0 ≠ 02 then
            if not MisalignedSupport()
                then SignalException(AddressError)
                else /* implementation dependent misalignment handling */
            endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
StoreFPR(ft, UNINTERPRETED_WORD,
        memdoubleword31+8*bytesel..8*bytesel)
```

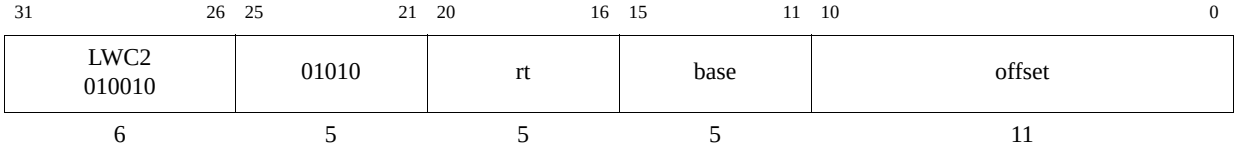
Exceptions:

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

MIPS32, (all release levels less than Release 6)



MIPS32 Release 6 (Release 6 and future)



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

Purpose: Load Word to Coprocessor 2

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 signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

Prior to MIPS32 Release 6: An Address Error exception occurs if $EffectiveAddress_{1..0} \neq 0$ (not word-aligned).

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

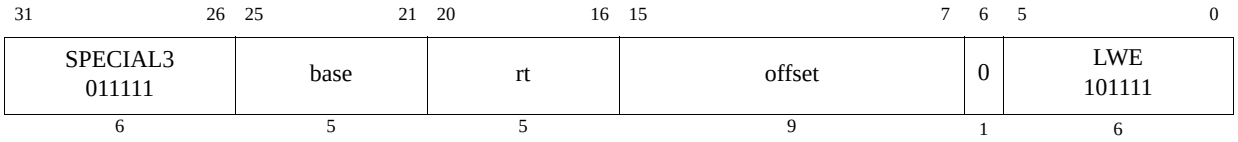
```
        vAddr ← sign_extend(offset) + GPR[base]
        if vAddr12..0 ≠ 02 then
           if not MisalignedSupport()
              then SignalException(AddressError)
              else /* implementation dependent misalignment handling */
           endif
        endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
CPR[2,rt,0] ← sign_extend(memdoubleword31+8*bytesel..8*bytesel)
```

Exceptions:

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

Programming Notes:

As shown in the instruction drawing above, the MIPS Release 6 architecture implements an 11-bit offset, whereas all release levels lower than Release 6 of the MIPS architecture implement a 16-bit offset.



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

Purpose: Load Word EVA

To load a word from user mode virtual address space when executing in kernel mode.

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

The LWE instruction functions in exactly the same fashion as the LW instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Restrictions:

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Prior to MIPS32 Release 6: 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.

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   if not MisalignedSupport()
      then SignalException(AddressError)
      else /* implementation dependent misalignment handling */
   endif
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)
```

Exceptions:

TLB Refill

TLB Invalid

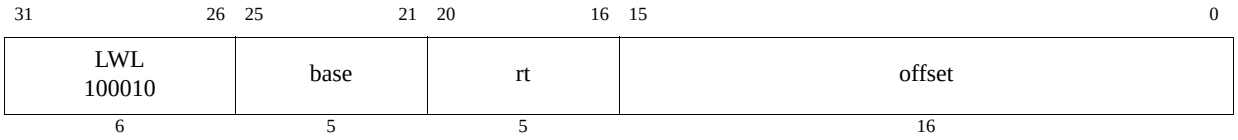
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



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

Purpose: Load Word Left

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]`

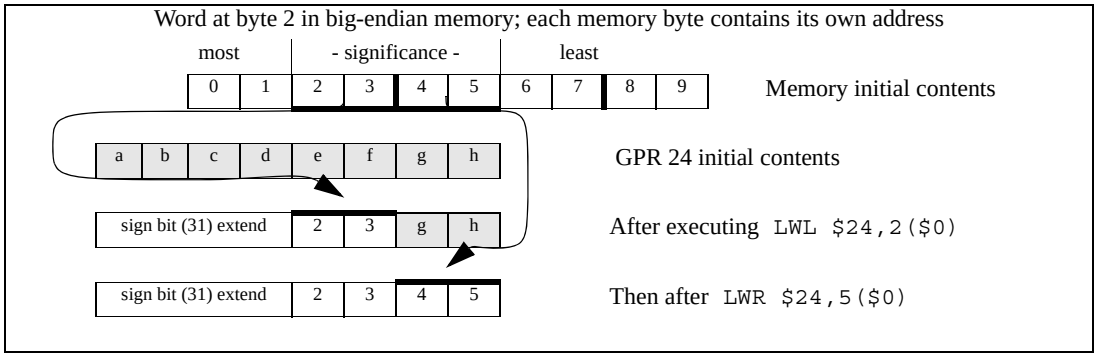
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

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.

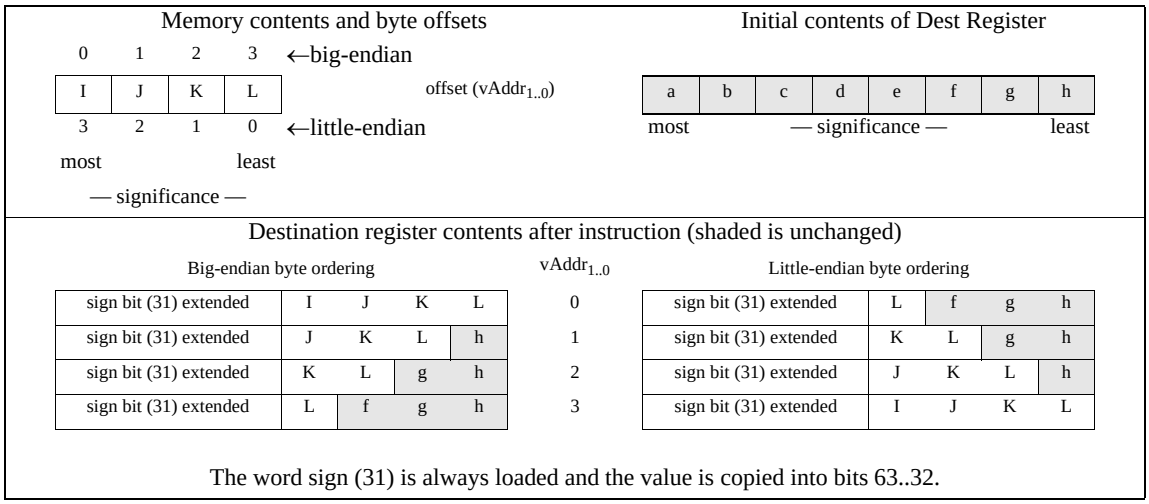
The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2 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 3.17 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 LWL Instruction



Restrictions:

None

Availability:

Release 6 removes the load/store-left/right family of instructions, and requires the system to support misaligned memory accesses.

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

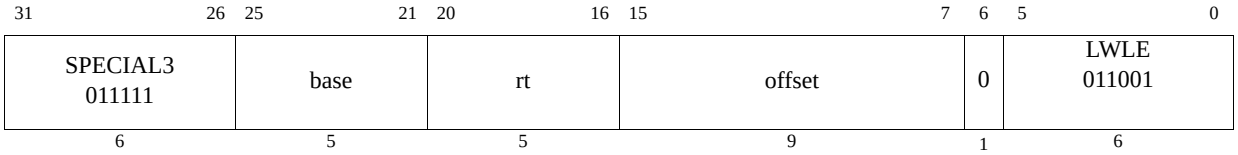
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

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.



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

Purpose: Load Word Left EVA

To load the most-significant part of a word as a signed value from an unaligned user mode virtual address while executing in kernel mode.

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

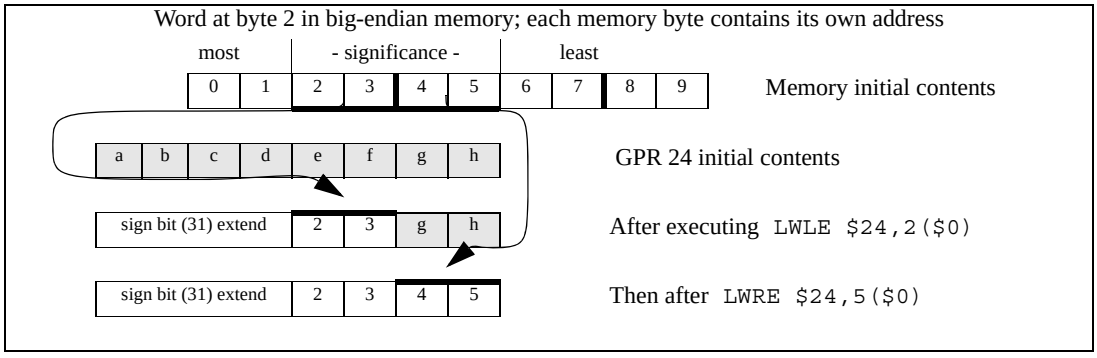
The 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

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, LWLE 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 LWRE loads the remainder of the unaligned word

Figure 3.19 Unaligned Word Load Using LWLE and LWRE

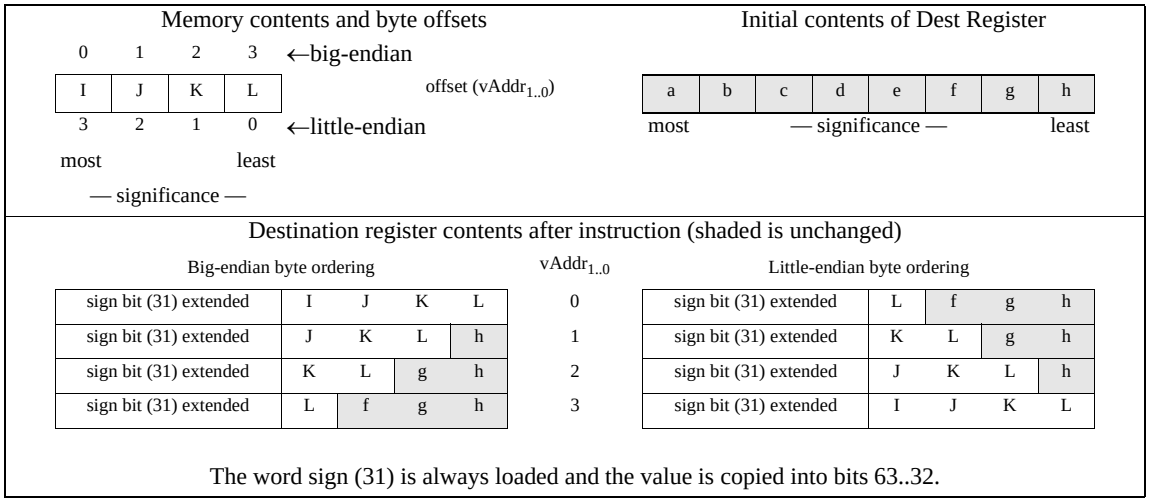


The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address ($vAddr_{1..0}$), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.

The LWLE instruction functions in exactly the same fashion as the LWL instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the *Config5*$_{EVA}$ field being set to one.

Figure 3.20 Bytes Loaded by LWLE Instruction



Restrictions:

This instruction has been removed in the MIPS32 Release 6 architecture.

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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:

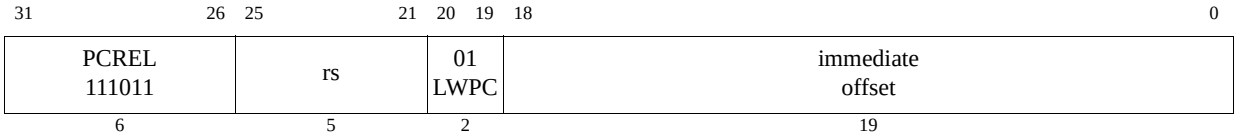
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch, Reserved Instruction, Coprocessor Unusable

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.



Format: `LWPC`
`LWPC rs, offset19` **MIPS32 Release 6**

Purpose: Load Word PC-relative

To load a word from memory as a signed value, using a PC-relative address.

Description: `GPR[rs] ← memory[ PC + sign_extend( offset << 2 ) ]`

The 19 bit offset is shifted left by 2 bits, sign-extended, and added to the address of the LWPC instruction.

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*.

LWPC, like all memory references, applies `memory_address` to the virtual address, sign extending if User mode and Status.UX=0.**Restrictions:**

None.

LWPC is naturally aligned, by specification.

Availability:

LWPC is introduced by and required as of MIPS32 Release 6.

Operation

```
vAddr ← memory_address( PC + sign_extend(offset)<<2 )
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← sign_extend(memword)
```

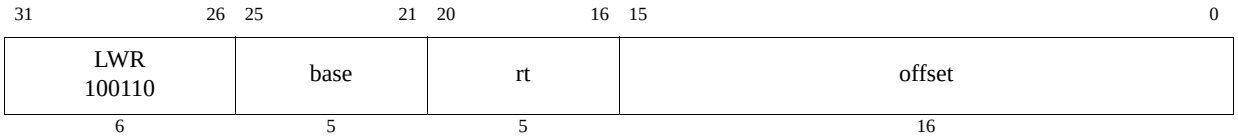
Exceptions:

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

Programming Note

The MIPS32 Release 6 PC-relative loads (LWPC, LWUPC, LDPC) are considered data references.

For the purposes of watchpoints (provided by the CP0 *WatchHi* and *WatchLo* registers) and EJTAG breakpoints, the PC-relative reference is considered to be a data reference, rather than an instruction reference. That is, the watchpoint or breakpoint is triggered only if enabled for data references.



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

Purpose: Load Word Right

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.

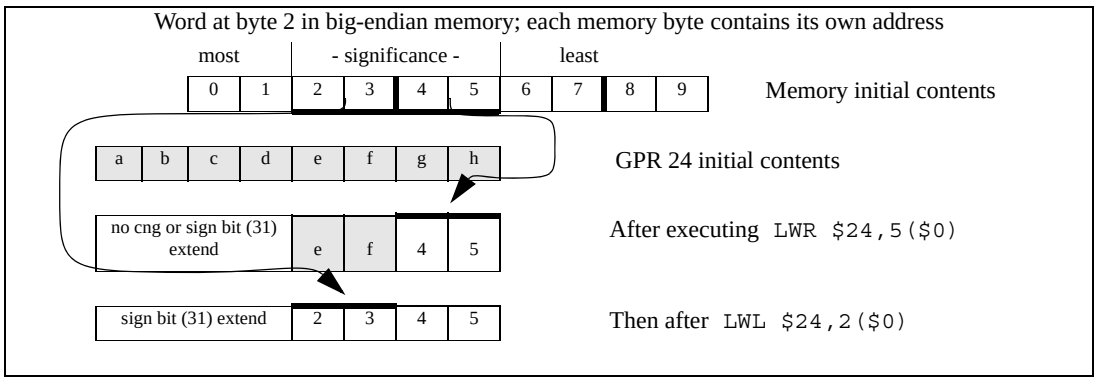
A part of *W*, the least-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. This part of *W* is loaded into the least-significant (right) part of the word in GPR *rt*. The remaining most-significant part of the word in GPR *rt* is unchanged.

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.

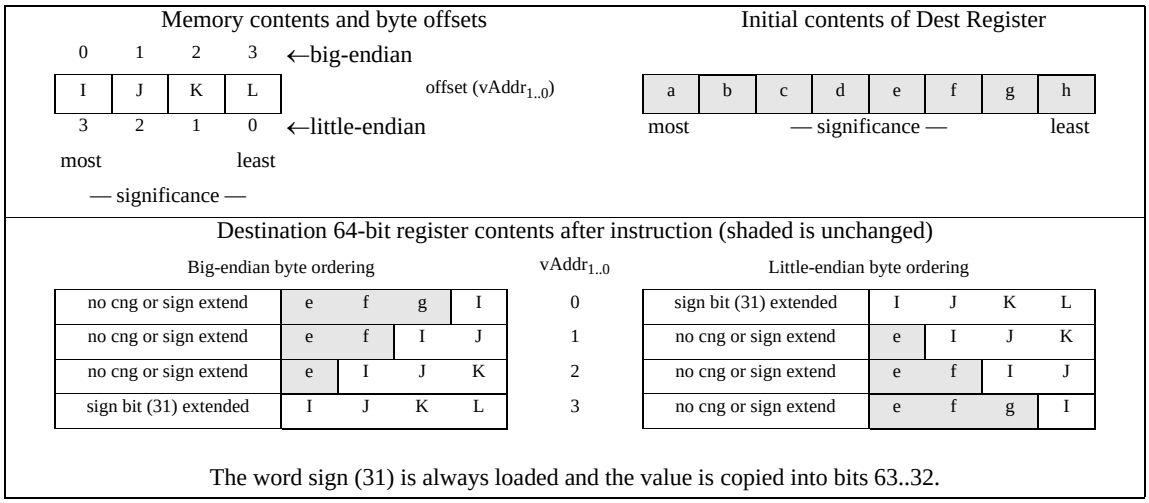
The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the least-significant byte at 5. First, LWR loads these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

Figure 3.21 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 3.22 Bytes Loaded by LWR Instruction



Restrictions:

None

Availability:

Release 6 removes the load/store-left/right family of instructions, and requires the system to support misaligned memory accesses.

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:

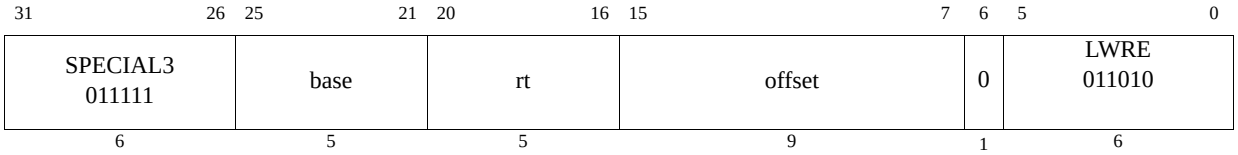
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

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.



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

Purpose: Load Word Right EVA

To load the least-significant part of a word from an unaligned user mode virtual memory address as a signed value while executing in kernel mode.

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

The 9-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.

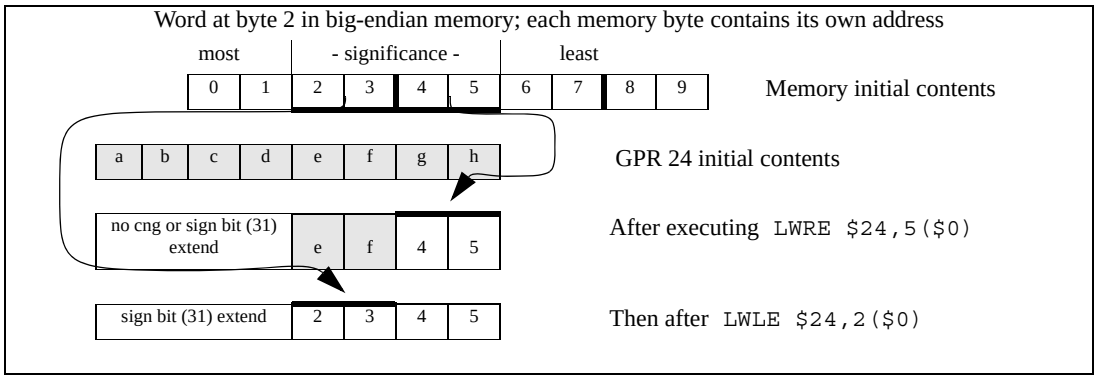
Executing both LWRE and LWLE, in either order, delivers a sign-extended word value in the destination register.

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, LWRE loads these 2 bytes into the right part of the destination register. Next, the complementary LWLE loads the remainder of the unaligned word.

The LWRE instruction functions in exactly the same fashion as the LWR instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

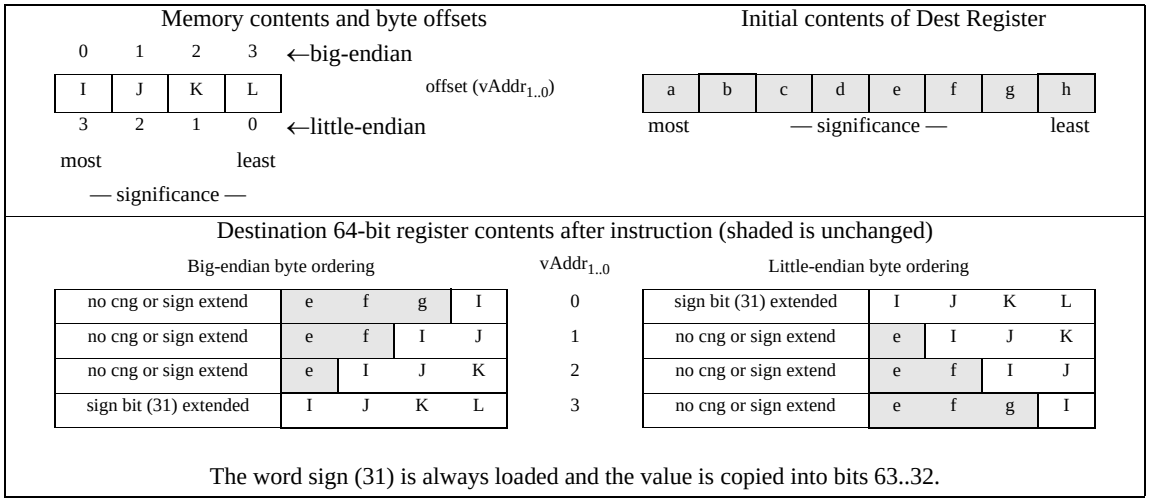
Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Figure 3.23 Unaligned Word Load Using LWLE and LWRE



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 3.24 Bytes Loaded by LWRE Instruction



Restrictions:

This instruction has been removed in the MIPS32 Release 6 architecture.

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 ← (temp31)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:

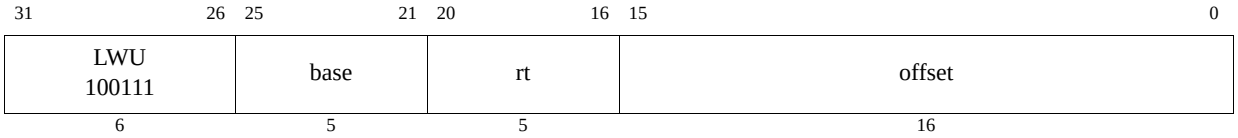
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch, Reserved Instruction, Coprocessor Unusable

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.



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

Purpose: Load Word Unsigned

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:

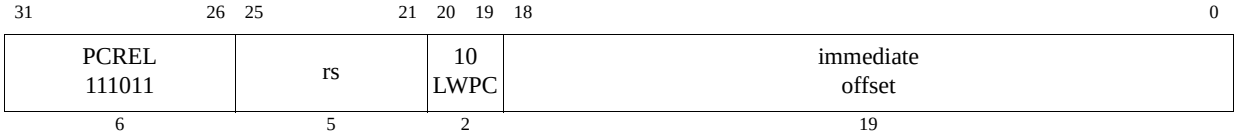
The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
   if vAddr1..0 ≠ 0^2 then
      if not MisalignedSupport()
         then SignalException(AddressError)
         else /* implementation dependent misalignment handling */
      endif
   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] ← 0^32 || memdoublewordPSIZE-1+8*byte..8*byte
```

Exceptions:

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



Format: `LWUPC`
`LWUPC rs, offset19` **MIPS32 Release 6**

Purpose: Load Word Unsigned PC-relative

To load a word from memory as an unsigned value, using a PC-relative address.

Description: `GPR[rs] ← memory[ PC + sign_extend( imm19 << 2 ) ]`

The 19 bit immediate is shifted left by 2 bits, sign-extended, and added to the address of the LWUPC instruction.

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, zero-extended to the GPR register length if necessary, and placed in GPR *rt*.

LWUPC, like all memory references, applies `memory_address` to the virtual address, sign extending if User mode and Status.UX=0.

Restrictions:

None.

LWUPC is naturally aligned, by specification.

Availability:

LWUPC is required as of MIPS32 Release 6.

Operation[1]

```
vAddr ← memory_address( PC + sign_extend(offset)<<2 )
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← zero_extend(memword)
```

Exceptions:

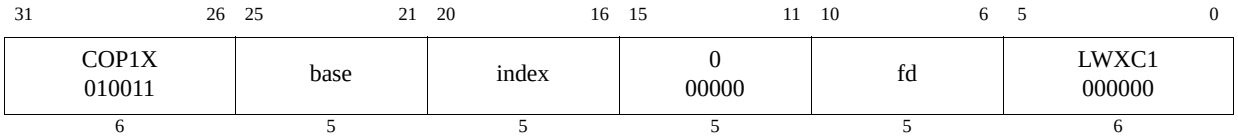
TLB Refill, TLB Invalid, TLB Read Inhibit, Bus Error, Address Error, Watch

Programming Note

The MIPS32 Release 6 PC-relative loads (LWPC, LWUPC, LDPC) are considered data references.

For the purposes of watchpoints (provided by the CP0 *WatchHi* and *WatchLo* registers) and EJTAG breakpoints, the PC-relative reference is considered to be a data reference, rather than an instruction reference. That is, the watchpoint or breakpoint is triggered only if enabled for data references.

1. This pseudocode is inherited from LW. It does not reflect the handling of memory alignment and endianness given MIPS32 Release 6's support for misaligned memory accesses.



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

Purpose: Load Word Indexed to Floating Point

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*. If FPRs are 64 bits wide, bits 63..32 of FPR *fs* become **UNPREDICTABLE**. The contents of GPR *index* and GPR *base* are added to form the effective address.

Restrictions:

This instruction has been removed[1] in the MIPS32 Release 6 architecture.

Prior to MIPS32 Release 6, the following restrictions apply:

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

Availability:

Removed by Release 6.

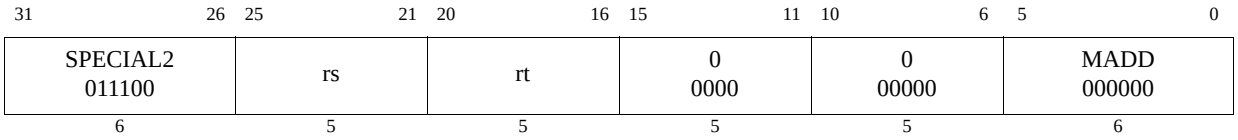
Operation:

```
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(fd, UNINTERPRETED_WORD,
          memdoubleword31+8*bytesel..8*bytesel)
```

Exceptions:

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

1. "Removed by MIPS32 Release 6" means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Multiply and Add Word to Hi,Lo

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

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)
```

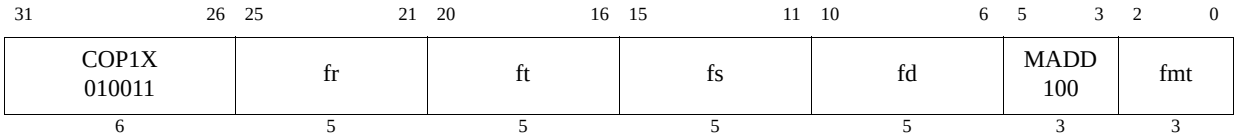
Exceptions:

None

For implementations that do not implement the DSP Module, it is recommended that this instruction should cause a Reserved Instruction Exception if bits 15:11 are not zero. This is to enable emulation of DSP Module instructions.

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.fmt`

`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: Floating Point Multiply Add

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 intermediate product is rounded according to the current rounding mode in *FCSR*. The value in FPR *fr* is added to the product. The result sum is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. The results and flags are as if separate floating-point multiply and add instructions were executed.

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 the *FR*=0 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Compatibility and Availability:

MADD.S and MADD.D: Required in all versions of MIPS64 since MIPS64 Release 1. Not available in MIPS32 Release1. Required by MIPS32 Release 2 and subsequent versions of MIPS32.

Required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (FIR_{F64}=0 or 1, $Status_{FR}$=0 or 1).

This instruction has been removed by the MIPS32 Release 6 architecture and has been replaced by the fused multiply-add instruction. Refer to the fused multiply-add instruction ‘MADDF.fmt’ in this manual for more information. Note that MIPS32 Release 6 does not support Paired Single (PS).

Operation:

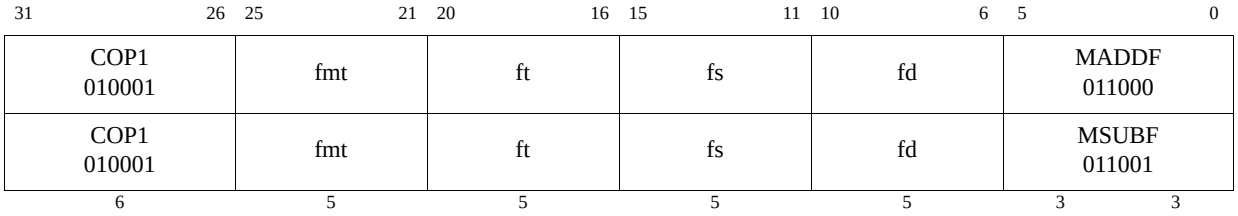
```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, (vfs ×fmt vft) +fmt vfr)
```

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
COP1 010001	fmt	ft	fs	fd	MADDF 011000
COP1 010001	fmt	ft	fs	fd	MSUBF 011001
6	5	5	5	5	3 3

Format: `MADDF.fmt MSUBF.fmt`

`MADDF.S fd, fs, ft` **MIPS32 Release 6**

`MADDF.D fd, fs, ft` **MIPS32 Release 6**

`MSUBF.S fd, fs, ft` **MIPS32 Release 6**

`MSUBF.D fd, fs, ft` **MIPS32 Release 6**

Purpose: Floating Point Fused Multiply Add, Floating Point Fused Multiply Subtract

MADDF.fmt: To perform a fused multiply-add of FP values

MSUBF.fmt: To perform a fused multiply-subtract of FP values

Description:

```
MADDF.fmt: FPR[fd] ← FPR[fd] + (FPR[fs] × FPR[ft])
MSUBF.fmt: FPR[fd] ← FPR[fd] - (FPR[fs] × FPR[ft])
```

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce an intermediate product. The intermediate product is calculated to infinite precision. The product is added to the value in FPR *fd*. 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*.

(For MSUB.fmt, the product is subtracted from the value in FPR *fd*.)

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

Restrictions:

Compatibility and Availability:

MADDF.fmt and MSUBF.fmt are required in MIPS32 Release 6.

MADDF.fmt and MSUBF.fmt are not available in architecture releases less than MIPS32 Release 6.

The fused multiply add instructions, MADDF.fmt and MSUBF.fmt, replace instructions pre-MIPS32 Release 6 such as MADD.fmt, MSUB.fmt, NMADD.fmt, and NMSUB.fmt. In MIPS32 Release 5 the replaced instructions were unfused multiply-add, with an intermediate rounding, although in some earlier implementations they were fused: i.e. instructions such as MADD.fmt were usually unfused, but were occasionally fused. MIPS32 Release 6 provides consistent behavior: MADDF.fmt and MSUBF.fmt are required to be fused in all implementations.

MIPS32 Release 6 MSUBF.fmt, `fd←fd-fs×ft`, corresponds more closely to pre-MIPS32 Release 6 NMADD.fmt, `fd←fr-fs×ft`, than to pre-MIPS32 Release 6 MSUB.fmt, `fd←fs×ft-fr`.

FPU scalar MADDF.fmt corresponds to MSA vector MADD.df.

FPU scalar MSUBF.fmt corresponds to MSA vector MSUB.df.

Operation:

```
ValidateAccessToFPUResources(fmt,{S,D})
vfr ← ValueFPR(fr, fmt)
```

```
vfs ← ValueFPR(fs, fmt)
vfd ← ValueFPR(fd, fmt)
MADDF.fmt: vinf ← vfd +∞ (vfs *∞ vft)
MADDF.fmt: vinf ← vfd -∞ (vfs *∞ vft)
StoreFPR(fd, fmt, vinf)
```

Special Considerations:

The fused multiply-add computation is performed in infinite precision, and signals Inexact, Overflow, or Underflow if and only if the final result differs from the infinite precision result in the appropriate manner.

Like most FPU computational instructions, if the flush-subnormals-to-zero mode, FCSR.FS=1, then subnormals are flushed before beginning the fused-multiply-add computation, and Inexact may be signaled.

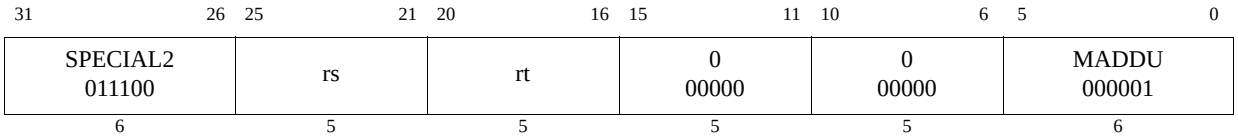
I.e. Inexact may be signaled both by input flushing and/or by the fused-multiply-add: the conditions or ORed.

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow



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

Purpose: Multiply and Add Unsigned Word to Hi,Lo

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

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)
```

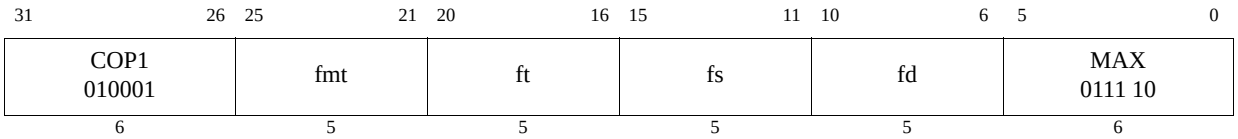
Exceptions:

None

For implementations that do not implement the DSP Module, it is recommended that this instruction should cause a Reserved Instruction Exception if bits 15:11 are not zero. This is to enable emulation of DSP Module instructions.

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: `MAX/MIN/MAXA/MINA .fmt family`

`MAX.fmt, , ,`

`MAX.S fd,fs,ft` **MIPS32 Release 6**

`MAX.D fd,fs,ft` **MIPS32 Release 6**

Purpose: Scalar Floating-Point Max/Min/maxNumMag/minNumMag

Scalar Floating-Point Maximum

Description:

`MAX.fmt: FPR[fd]← maxNum(FPR[fs],FPR[ft])`

MAX.fmt writes the maximum value of the inputs `fs` and `ft` to the destination `fd`

The instructions MAX.fmt/MIN.fmt/MAXA.fmt/MINA.fmt correspond to the IEEE 754-2008 operations maxNum/minNum/maxNumMag/minNumMag.

MAX.fmt corresponds to the IEEE 754-2008 operation maxNum.

Numbers are preferred to NaNs: if one input is a NaN, but not both, the value of the numeric input is returned. (If both are NaNs, the NaN in `fs` is returned.)

The scalar FPU instructions MAX.fmt/MIN.fmt/MAXA.fmt/MINA.fmt correspond to the MSA instructions FMAX.df/FMIN.df/FMAXA.df/FMINA.df

Scalar FPU instruction MAX.fmt corresponds to the MSA vector instruction FMAX.df.

Restrictions:

Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008. See also the section “Special Cases”, below.

Operation:

```
if not IsCoprocessorEnabled(1)
   then SignalException(CoprocessorUnusable, 1)endif
if not IsFloatingPointImplemented(fmt)
   then SignalException(ReservedInstruction)endif

if SNan(ValueFPR(fs,fmt)) or SNan(ValueFPR(ft,fmt))
   then SignalException(InvalidOperand) endif

if Nan(ValueFPR(fs,fmt)) and Nan(ValueFPR(ft,fmt))then
   StoreFPR(fd, fmt, ValueFPR(fs,fmt))
elseif Nan(ValueFPR(fs,fmt)) then
   StoreFPR (fd, fmt, ValueFPR(ft,fmt))
elseif Nan(ValueFPR(ft,fmt)) then
```

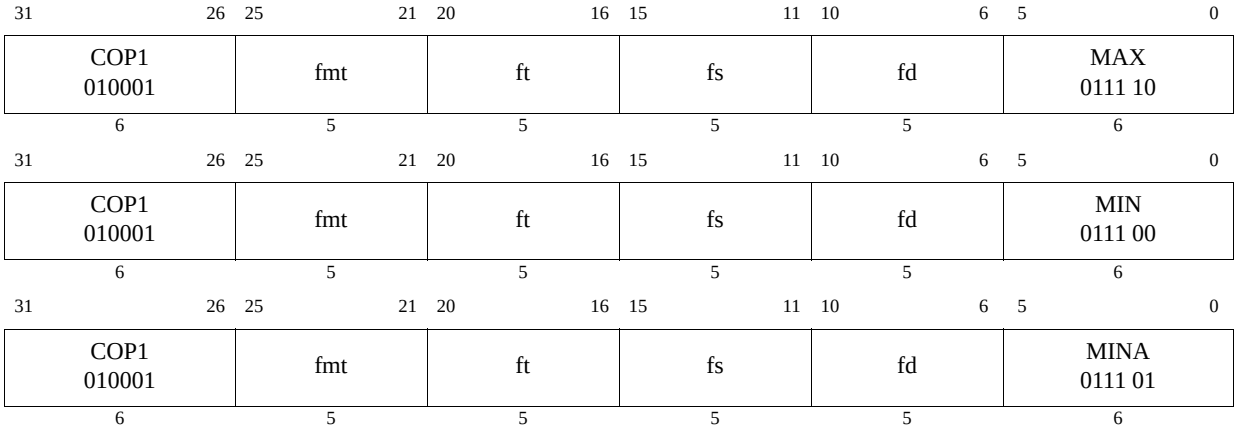
```
    StoreFPR (fd, fmt, ValueFPR(fs,fmt))
else
    case instruction of
    FMAX.fmt:   ftmp ← MaxFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
    end case

    StoreFPR (fd, fmt, ftmp)
endif

?/* end of instruction */

function MaxFP(tt, ts, n)
    /* Returns the largest argument. */
endfunction MaxFP
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow



Format: `MAX/MIN/MAXA/MINA .fmt family`
`MAX.fmt, , MIN.fmt, MINA.fmt`

```
MAX.S fd,fs,ft                                  MIPS32 Release 6
MAX.D fd,fs,ft                                  MIPS32 Release 6

MIN.S fd,fs,ft                                  MIPS32 Release 6
MIN.D fd,fs,ft                                  MIPS32 Release 6
MINA.S fd,fs,ft                                 MIPS32 Release 6
MIN_S.D fd,fs,ft                                MIPS32 Release 6
```

Purpose: Scalar Floating-Point Max/Min/maxNumMag/minNumMag

Scalar Floating-Point Maximum

Scalar Floating-Point Minimum

Scalar Floating-Point argument with Minimum Absolute Value

Description:

```
MAX.fmt:   FPR[fd]← maxNum(FPR[fs],FPR[ft])
MIN.fmt:   FPR[fd]← minNum(FPR[fs],FPR[ft])

MINA.fmt:   FPR[fd]← minNumMag(FPR[fs],FPR[ft])
```

MAX.fmt writes the maximum value of the inputs `fs` and `ft` to the destination `fd`

MIN.fmt writes the minimum value of the inputs `fs` and `ft` to the destination `fd`.

MINA.fmt takes input arguments `fs` and `ft` and writes the argument with the minimum absolute value to the destination `fd`.

The instructions MAX.fmt/MIN.fmt/MAXA.fmt/MINA.fmt correspond to the IEEE 754-2008 operations maxNum/minNum/maxNumMag/minNumMag.

MAX.fmt corresponds to the IEEE 754-2008 operation maxNum.

MIN.fmt corresponds to the IEEE 754-2008 operation minNum.

MINA.fmt corresponds to the IEEE 754-2008 operation minNumMag.

Numbers are preferred to NaNs: if one input is a NaN, but not both, the value of the numeric input is returned. (If both are NaNs, the NaN in `fs` is returned.)

The scalar FPU instructions MAX.fmt/MIN.fmt/MAXA.fmt/MINA.fmt correspond to the MSA instructions FMAX.df/FMIN.df/FMAXA.df/FMINA.df

Scalar FPU instruction MAX.fmt corresponds to the MSA vector instruction FMAX.df.

Scalar FPU instruction MIN.fmt corresponds to the MSA vector instruction FMIN.df.

Scalar FPU instruction MINA.fmt corresponds to the MSA vector instruction FMIN_A.df.

Restrictions:

Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008. See also the section “Special Cases”, below.

Operation:

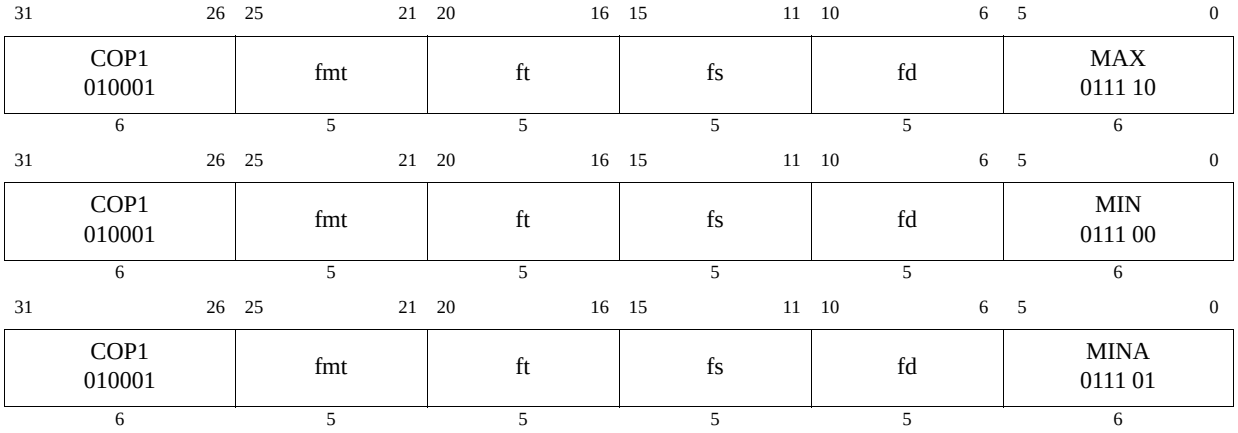
```
if SNan(ValueFPR(fs,fmt)) or SNan(ValueFPR(ft,fmt))
   then SignalException(InvalidOperand) endif

if Nan(ValueFPR(fs,fmt)) and Nan(ValueFPR(ft,fmt))then
   StoreFPR(fd, fmt, ValueFPR(fs,fmt))
elseif Nan(ValueFPR(fs,fmt)) then
   StoreFPR (fd, fmt, ValueFPR(ft,fmt))
elseif Nan(ValueFPR(ft,fmt)) then
   StoreFPR (fd, fmt, ValueFPR(fs,fmt))
else
   case instruction of
   FMAX.fmt:   ftmp ← MaxFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
   FMIN.fmt:   ftmp ← MinFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
   FMINA.fmt:  ftmp ← MinAbsoluteFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
   end case

   StoreFPR (fd, fmt, ftmp)
endif

?/* end of instruction */

function MaxFP(tt, ts, n)
   /* Returns the largest argument. */
endfunction MaxFP

function MinFP(tt, ts, n)
   /* Returns the smallest argument. */
endfunction MaxFP


function MinAbsoluteFP(tt, ts, n)
   /*
      For equal absolute values, returns the smallest positive argument.*/
endfunction MinAbsoluteFP
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow



Format: `MAX/MIN/MAXA/MINA .fmt family`
`MAX.fmt, , MIN.fmt, MINA.fmt`

`MAX.S fd,fs,ft` **MIPS32 Release 6**
`MAX.D fd,fs,ft` **MIPS32 Release 6**

`MIN.S fd,fs,ft` **MIPS32 Release 6**
`MIN.D fd,fs,ft` **MIPS32 Release 6**
`MINA.S fd,fs,ft` **MIPS32 Release 6**
`MIN_S.D fd,fs,ft` **MIPS32 Release 6**

Purpose: Scalar Floating-Point Max/Min/maxNumMag/minNumMag

Scalar Floating-Point Maximum

Scalar Floating-Point Minimum

Scalar Floating-Point argument with Minimum Absolute Value

Description:

```
MAX.fmt:   FPR[fd]← maxNum(FPR[fs],FPR[ft])
MIN.fmt:   FPR[fd]← minNum(FPR[fs],FPR[ft])

MINA.fmt:   FPR[fd]← minNumMag(FPR[fs],FPR[ft])
```

MAX.fmt writes the maximum value of the inputs `fs` and `ft` to the destination `fd`

MIN.fmt writes the minimum value of the inputs `fs` and `ft` to the destination `fd`.

MINA.fmt takes input arguments `fs` and `ft` and writes the argument with the minimum absolute value to the destination `fd`.

The instructions MAX.fmt/MIN.fmt/MAXA.fmt/MINA.fmt correspond to the IEEE 754-2008 operations maxNum/minNum/maxNumMag/minNumMag.

MAX.fmt corresponds to the IEEE 754-2008 operation maxNum.

MIN.fmt corresponds to the IEEE 754-2008 operation minNum.

MINA.fmt corresponds to the IEEE 754-2008 operation minNumMag.

Numbers are preferred to NaNs: if one input is a NaN, but not both, the value of the numeric input is returned. (If both are NaNs, the NaN in `fs` is returned.)

The scalar FPU instructions MAX.fmt/MIN.fmt/MAXA.fmt/MINA.fmt correspond to the MSA instructions FMAX.df/FMIN.df/FMAXA.df/FMINA.df

Scalar FPU instruction MAX.fmt corresponds to the MSA vector instruction FMAX.df.

Scalar FPU instruction MIN.fmt corresponds to the MSA vector instruction FMIN.df.

Scalar FPU instruction MINA.fmt corresponds to the MSA vector instruction FMIN_A.df.

Restrictions:

Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008. See also the section “Special Cases”, below.

Operation:

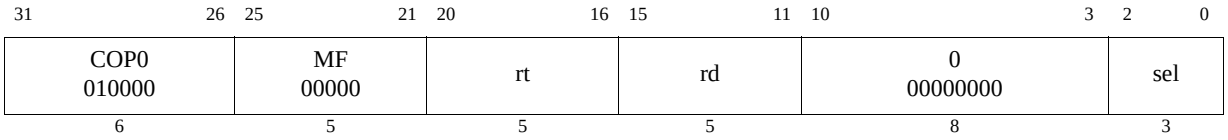
```
if not IsCoprocessorEnabled(1)
   then SignalException(CoprocessorUnusable, 1)endif
if not IsFloatingPointImplemented(fmt)
   then SignalException(ReservedInstruction)endif

if SNan(ValueFPR(fs,fmt)) or SNan(ValueFPR(ft,fmt))
   then SignalException(InvalidOperand) endif

if Nan(ValueFPR(fs,fmt)) and Nan(ValueFPR(ft,fmt))then
   StoreFPR(fd, fmt, ValueFPR(fs,fmt))
elseif Nan(ValueFPR(fs,fmt)) then
   StoreFPR (fd, fmt, ValueFPR(ft,fmt))
elseif Nan(ValueFPR(ft,fmt)) then
   StoreFPR (fd, fmt, ValueFPR(fs,fmt))
else
   case instruction of
   FMAX.fmt:   ftmp ← MaxFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
   FMIN.fmt:   ftmp ← MinFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
   FMINA.fmt:  ftmp ← MinAbsoluteFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
   end case

   StoreFPR (fd, fmt, ftmp)
endif

?/* end of instruction */

function MaxFP(tt, ts, n)
   /* Returns the largest argument. */
endfunction MaxFP

function MinFP(tt, ts, n)
   /* Returns the smallest argument. */
endfunction MaxFP


function MinAbsoluteFP(tt, ts, n)
   /*
       For equal absolute values, returns the smallest positive argument.*/
endfunction MinAbsoluteFP
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow



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

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

Purpose: Move from Coprocessor 0

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.

When the coprocessor 0 register specified is the *EntryLo0* or the *EntryLo1* register, the RI/XI fields are moved to bits 31:30 of the destination register. This feature supports MIPS32 backward compatability on a MIPS64 system.

Restrictions:

The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *rd* and *sel*.

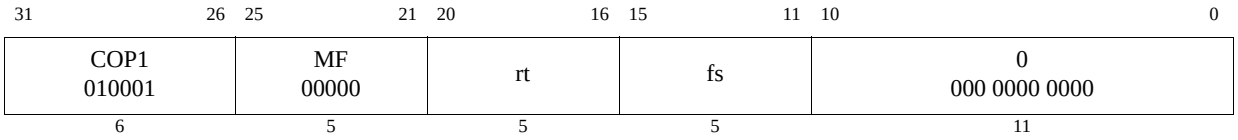
Operation:

```
reg = rd
data ← CPR[0,reg,sel]31..0
if (reg,sel = EntryLo1 or reg,sel = EntryLo0 then
   GPR[rt]29..0 ← data29..0
   GPR[rt]31 ← data63
   GPR[rt]30 ← data62
   GPR[rt]63..32 ← sign_extend(data63)
else
   GPR[rt] ← sign_extend(data)
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction



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

Purpose: Move Word From Floating Point

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:

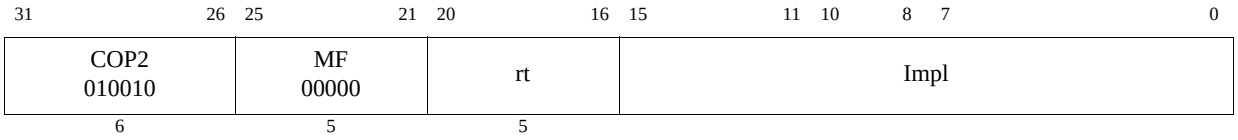
```
data ← ValueFPR(fs, UNINTERPRETED_WORD)31..0
GPR[rt] ← sign_extend(data)
```

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.



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

`MFC2, rt, Impl, sel` **MIPS32**

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

Purpose: Move Word From Coprocessor 2

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:

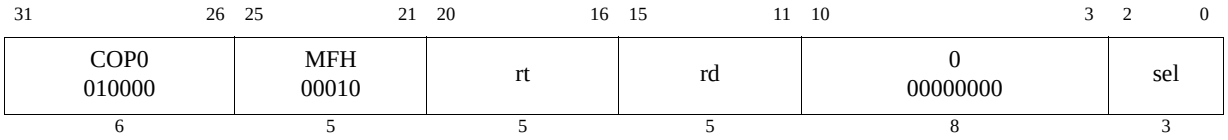
The results are **UNPREDICTABLE** if *Impl* specifies a coprocessor 2 register that does not exist.

Operation:

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

Exceptions:

Coprocessor Unusable



Format: `MFHC0 rt, rd` **MIPS32 Release 5**

`MFHC0 rt, rd, sel` **MIPS32 Release 5**

Purpose: Move from High Coprocessor 0

To move the contents of the upper 32 bits of a Coprocessor 0 register, extended by 32-bits, to a general register.

Description: `GPR[rt] ← CPR[0,rd,sel][63:32]`

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, and in those instances, the *sel* field must be zero.

When the Coprocessor 0 register specified is the *EntryLo0* or the *EntryLo1* register, MFHC0 must undo the effects of MTHC0, that is, bits 31:30 of the register must be returned as bits 1:0 of the GPR, and bits 32 and those of greater significance must be left-shifted by two and written to bits 31:2 of the GPR.

This feature supports MIPS32 backward-compatability of MIPS64 systems.

Restrictions:

The results are **UNDEFINED** if Coprocessor 0 does not contain a register as specified by *rd* and *sel*, or the register exists but is not extended by 32-bits, or the register is extended for XPA, but XPA is not supported or enabled.

Operation:

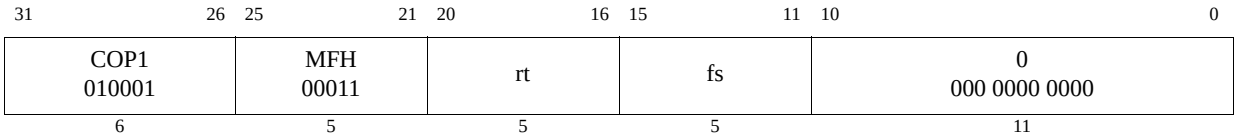
PABITS is the total number of physical address bits implemented. PABITS is defined in the descriptions of *EntryLo0* and *EntryLo1*.

```
reg ← rd
   data ← CPR[0,reg,sel]
   if (reg,sel = EntryLo1 or reg,sel = EntryLo0) then
      if (Config3LPA = 1 and PageGrainELPA = 1) then // PABITS > 36
         GPR[rt]31:0 ← data61..30
         GPR[rt]63..32 ← (data61)32 // sign-extend
      endif
   else
      GPR[rt] ← sign_extend(data63..32)
   endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction



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

Purpose: Move Word From High Half of Floating Point Register

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.

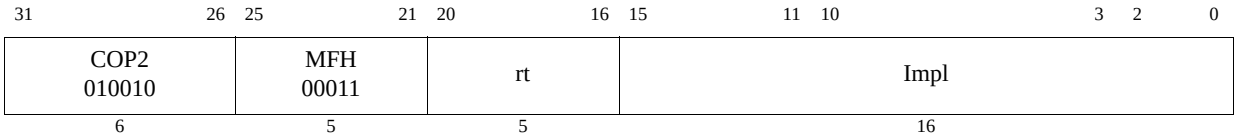
Operation:

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

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: `MFHC2 rt, Impl` **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: Move Word From High Half of Coprocessor 2 Register

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.

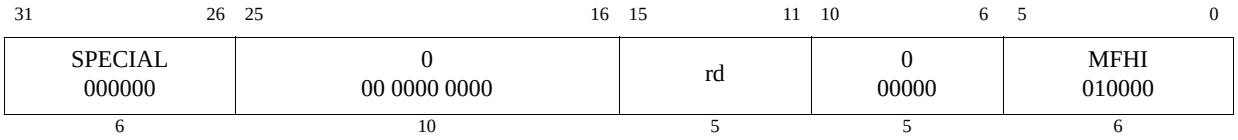
Operation:

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

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: `MFHI rd` **MIPS32**

Purpose: Move From HI Register

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

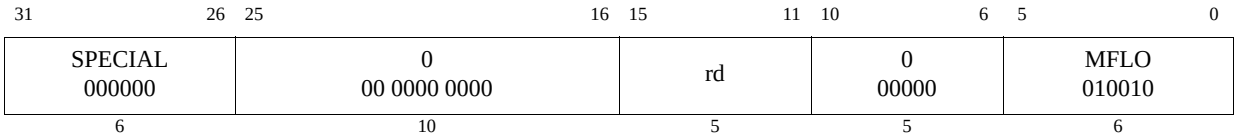
```
GPR[rd] ← HI
```

Exceptions:

None

Historical Information:

In the MIPS I, II, and III architectures, the two instructions which follow the MFHI must not modify 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: `MFLO rd` **MIPS32**

Purpose: Move From LO Register

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

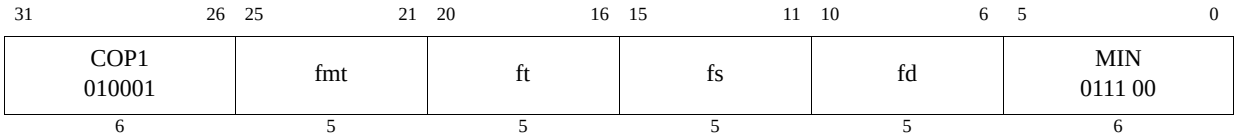
```
GPR[rd] ← LO
```

Exceptions:

None

Historical Information:

In the MIPS I, II, and III architectures, the two instructions which follow the MFLO must not modify the *HI* register. If this restriction is violated, the result of the MFLO is **UNPREDICTABLE**. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.



Format: `MIN.fmt`

`MIN.S fd,fs,ft` **MIPS32 Release 6**

`MIN.D fd,fs,ft` **MIPS32 Release 6**

Purpose: Scalar Floating-Point Minimum

Description:

`MIN.fmt: FPR[fd]← minNum(FPR[fs],FPR[ft])`

MIN.fmt writes the minimum value of the inputs `fs` and `ft` to the destination `fd`.

olute value to tMIN.fmt corresponds to the IEEE 754-2008 operation minNum.

Numbers are preferred to NaNs: if one input is a NaN, but not both, the value of the numeric input is returned. (If both are NaNs, the NaN in `fs` is returned.)

Scalar FPU instruction MIN.fmt corresponds to the MSA vector instruction FMIN.df.

Scalar FPU instruction MINA.fmt corresponds to the MSA vector instruction FMIN_A.df.

Restrictions:

Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008. See also the section “Special Cases”, below.

Operation:

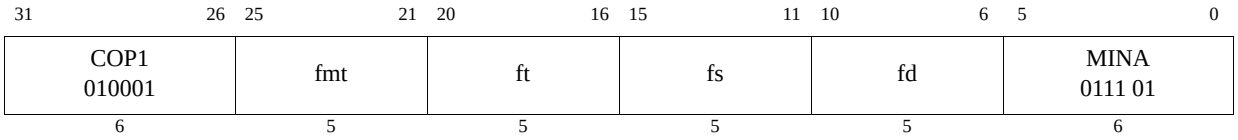
```
if SNan(ValueFPR(fs,fmt)) or SNan(ValueFPR(ft,fmt))
   then SignalException(InvalidOperand) endif

if Nan(ValueFPR(fs,fmt)) and Nan(ValueFPR(ft,fmt))then
   StoreFPR(fd, fmt, ValueFPR(fs,fmt))
elseif Nan(ValueFPR(fs,fmt)) then
   StoreFPR (fd, fmt, ValueFPR(ft,fmt))
elseif Nan(ValueFPR(ft,fmt)) then
   StoreFPR (fd, fmt, ValueFPR(fs,fmt))
else
   FMIN.fmt:   ftmp ← MinFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
   StoreFPR (fd, fmt, ftmp)
endif

?/* end of instruction */

function MinFP(tt, ts, n)
   /* Returns the smallest argument. */
endfunction MaxFP
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow



Format: `MINA.fmt`
`MINA.S fd,fs,ft` **MIPS32 Release 6**
`MIN_S.D fd,fs,ft` **MIPS32 Release 6**

Purpose: Scalar Floating-Point argument with Minimum Absolute Value

Description:

`MINA.fmt: FPR[fd]← minNumMag(FPR[fs],FPR[ft])`

MINA.fmt takes input arguments `fs` and `ft` and writes the argument with the minimum absoluteto the destination `fd`.

MINA.fmt corresponds to the IEEE 754-2008 operation minNumMag.

Numbers are preferred to NaNs: if one input is a NaN, but not both, the value of the numeric input is returned. (If both are NaNs, the NaN in `fs` is returned.)

Restrictions:

Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008. See also the section “Special Cases”, below.

Operation:

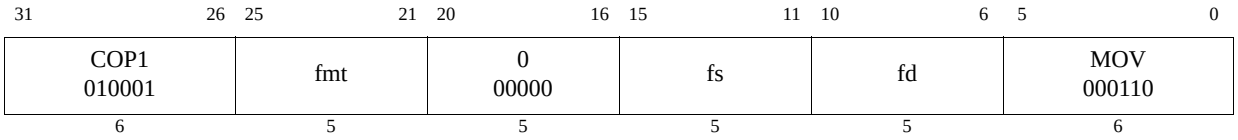
```
if SNan(ValueFPR(fs,fmt)) or SNan(ValueFPR(ft,fmt))
   then SignalException(InvalidOperand) endif

if Nan(ValueFPR(fs,fmt)) and Nan(ValueFPR(ft,fmt))then
   StoreFPR(fd, fmt, ValueFPR(fs,fmt))
elseif Nan(ValueFPR(fs,fmt)) then
   StoreFPR (fd, fmt, ValueFPR(ft,fmt))
elseif Nan(ValueFPR(ft,fmt)) then
   StoreFPR (fd, fmt, ValueFPR(fs,fmt))
else
   FMINA.fmt:  ftmp ← MinAbsoluteFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
   StoreFPR (fd, fmt, ftmp)
endif

?/* end of instruction */


function MinAbsoluteFP(tt, ts, n)
   /*
      For equal absolute values, returns the smallest positive argument.*/
endfunction MinAbsoluteFP
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow



Format: `MOV.fmt`

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

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

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

Purpose: Floating Point Move

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

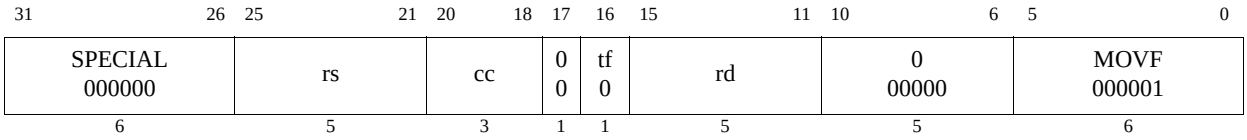
```
StoreFPR(fd, fmt, ValueFPR(fs, fmt))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation



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

Purpose: Move Conditional on Floating Point False

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:

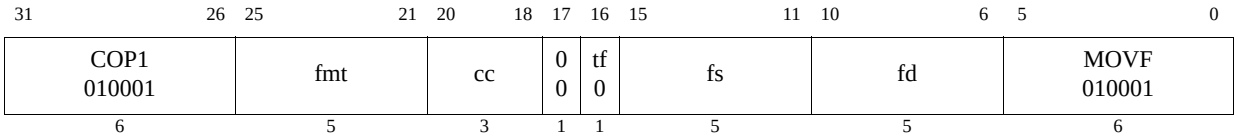
This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

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

Exceptions:

Reserved Instruction, Coprocessor Unusable



Format: `MOVF.fmt`

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

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

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

Purpose: Floating Point Move Conditional on Floating Point False

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

This instruction has been removed in the Release 6 architecture.

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(f, PS, ByteMerge(mask, f, fs))
endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

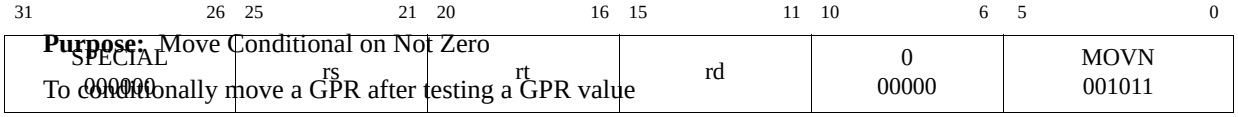
Unimplemented Operation

Programming Notes:

This instruction has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the 'SEL.fmt' instruction. Refer to the SEL.fmt instruction in this manual for more information. Note that MIPS32 Release 6 does not support Paired Single (PS).

1. "Removed by MIPS32 Release 6" means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.

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



Purpose: Move Conditional on Not Zero

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

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

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

Exceptions:

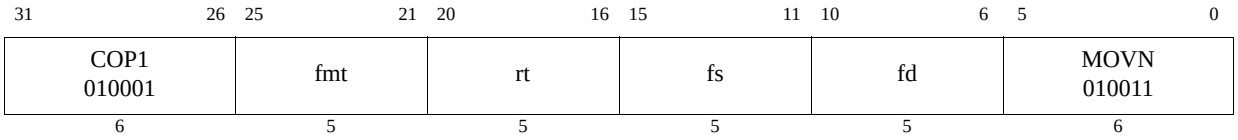
None

Programming Notes:

The non-zero value tested might be the *condition true* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions or a boolean value read from memory.

This instruction has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the ‘SELNEZ’ instruction. Refer to the SELNEZ instruction in this manual for more information.

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



Format: `MOVN.fmt`

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

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

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

Purpose: Floating Point Move Conditional on Not Zero

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

This instruction has been removed in the Release 6 architecture.

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:

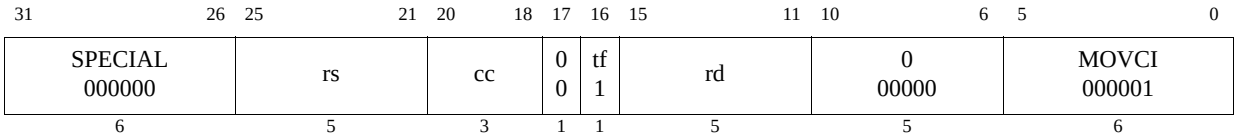
Unimplemented Operation

Programming Notes:

This instruction has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the 'SELNEZ.fmt' instruction. Refer to the SELNEZ.fmt instruction in this manual for more information. Note that

MIPS32 Release 6 does not support Paired Single (PS).

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Move Conditional on Floating Point True

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

Availability:

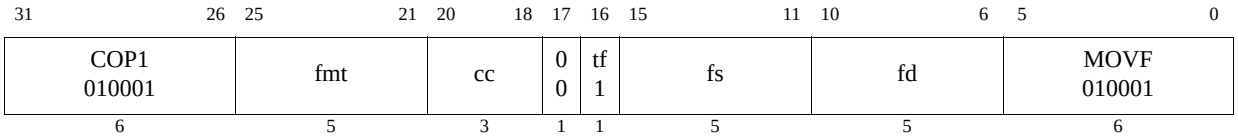
This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

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

Exceptions:

Reserved Instruction, Coprocessor Unusable



Format: `MOVT.fmt`

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

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

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

Purpose: Floating Point Move Conditional on Floating Point True

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 *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**.

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

This instruction has been removed in the Release 6 architecture.

Operation:

```
if fmt ≠ PS
   if FPConditionCode(cc) = 1 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(f, PS, ByteMerge(mask, f, fs))
endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

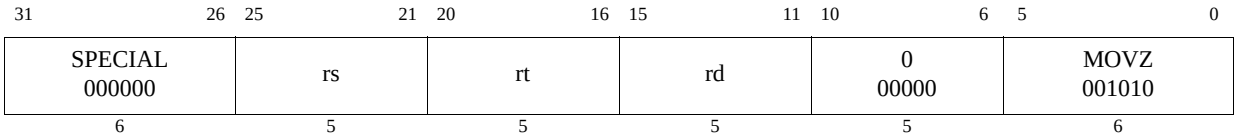
Floating Point Exceptions:

Unimplemented Operation

Programming Notes:

This instruction has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the 'SEL.fmt' instruction. Refer to the SEL.fmt instruction in this manual for more information. Note that MIPS32 Release 6 does not support Paired Single (PS).

1. "Removed by MIPS32 Release 6" means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Move Conditional on Zero

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

Availability:

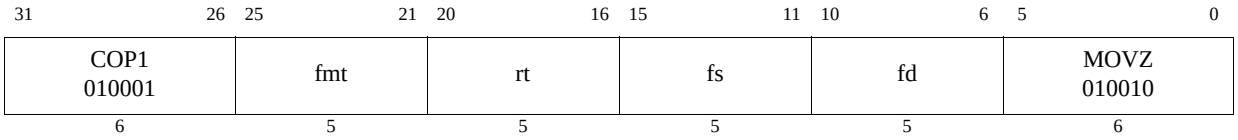
This instruction has been removed in the MIPS32 Release 6 architecture.

Programming Notes:

The zero value tested might be the *condition false* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions or a boolean value read from memory.

This instruction has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the ‘SELEQZ’ instruction. Refer to the SELEQZ instruction in this manual for more information.

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



Format: `MOVZ.fmt`

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

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

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

Purpose: Floating Point Move Conditional on Zero

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 *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes **UNPREDICTABLE**.

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

Restrictions:

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

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

The result of MOVZ.PS is **UNPREDICTABLE** if the processor is executing in the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

This instruction has been removed in the Release 6 architecture.

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:

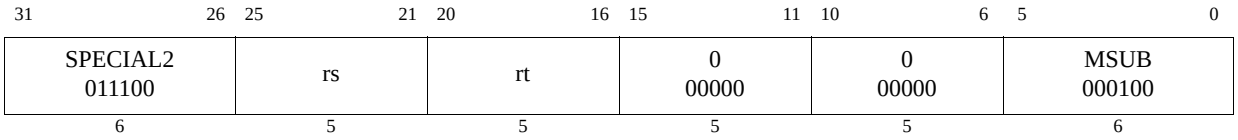
Unimplemented Operation

Programming Notes:

This instruction has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the ‘SELEQZ.fmt’ instruction. Refer to the SELEQZ.fmt instruction in this manual for more information. Note that

MIPS32 Release 6 does not support Paired Single (PS).

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Multiply and Subtract Word to Hi,Lo

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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)
```

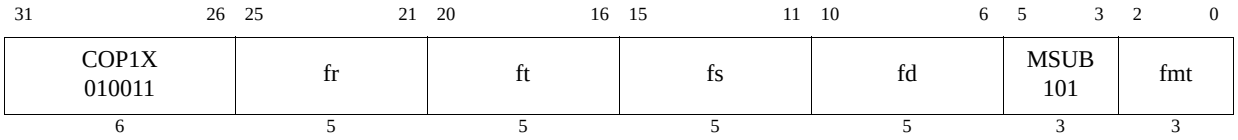
Exceptions:

None

For implementations that do not implement the DSP Module, it is recommended that this instruction should cause a Reserved Instruction Exception if bits 15:11 are not zero. This is to enable emulation of DSP Module instructions.

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.fmt`

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

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

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

Purpose: Floating Point Multiply Subtract

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 intermediate product is rounded according to the current rounding mode in *FCSR*. The subtraction result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. The results and flags are as if separate floating-point multiply and subtract instructions were executed.

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 the *FR*=0 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Compatibility and Availability:

MSUB.S and MSUB.D: Required in all versions of MIPS64 since MIPS64 Release 1. Not available in MIPS32 Release 1. Required by MIPS32 Release2 and subsequent versions of MIPS32.

Required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (FIR_{F64}=0 or 1, $Status_{FR}$=0 or 1).

This instruction has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the fused multiply-subtract instruction. Refer to the fused multiply-add instruction ‘MSUBF.fmt’ in this manual for more information. Note that MIPS32 Release 6 does not support Paired Single (PS).

Operation:

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

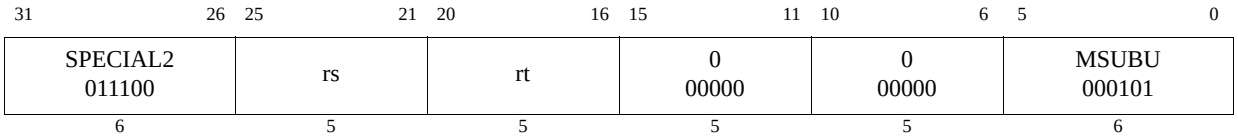
1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow



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

Purpose: Multiply and Subtract Word to Hi,Lo

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

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) - ((032 || GPR[rs]31..0) × (032 || GPR[rt]31..0))
HI ← sign_extend(temp63..32)
LO ← sign_extend(temp31..0)
```

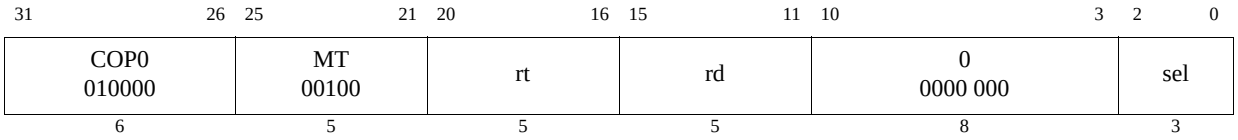
Exceptions:

None

For implementations that do not implement the DSP Module, it is recommended that this instruction should cause a Reserved Instruction Exception if bits 15:11 are not zero. This is to enable emulation of DSP Module instructions.

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

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

Purpose: Move to Coprocessor 0

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.

When the COP0 destination register specified is the *EntryLo0* or the *EntryLo1* register, bits 31:30 appear in the RI/XI fields of the destination register. This feature supports MIPS32 backward compatability on a MIPS64 system.

Restrictions:

The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *rd* and *sel*.

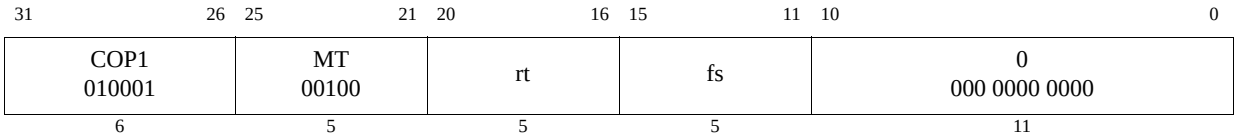
Operation:

```
data ← GPR[rt]
reg ← rd
   if (reg,sel = EntryLo1 or EntryLo0) then
      CPR[0,reg,sel]29..0 ← data29..0
      CPR[0,reg,sel]63 ← data31
      CPR[0,reg,sel]62 ← data30
      CPR[0,reg,sel]61:30 ← 032
   else if (Width(CPR[0,reg,sel]) = 64) then
      CPR[0,reg,sel] ← data
   else
      CPR[0,reg,sel] ← data31..0

endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction



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

Purpose: Move Word to Floating Point

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 **UNPREDICTABLE**.

Restrictions:

Operation:

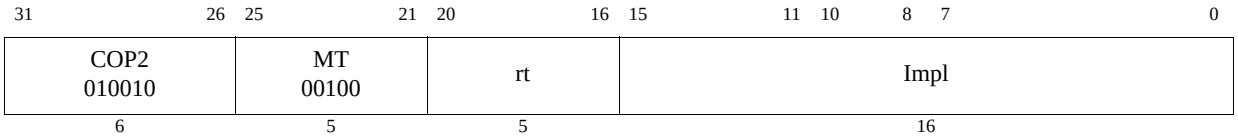
```
data ← GPR[rt]31..0
StoreFPR(fs, UNINTERPRETED_WORD, data)
```

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.



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

`MTC2 rt, Impl, sel` **MIPS32**

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

Purpose: Move Word to Coprocessor 2

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 a 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 **UNPREDICTABLE**. 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.

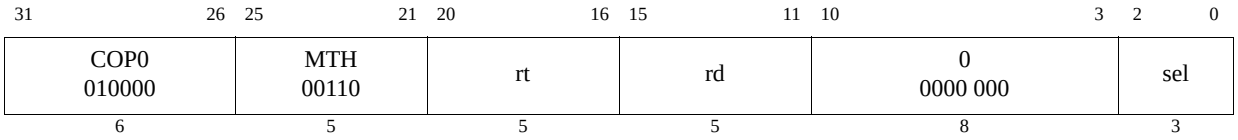
Operation:

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

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: `MTHC0 rt, rd` **MIPS32 Release 5**

`MTHC0 rt, rd, sel` **MIPS32 Release 5**

Purpose: Move to High Coprocessor 0

To copy a word from a GPR to the upper 32 bits of a COP2 general register that has been extended by 32 bits.

Description: `CPR[0, rd, sel][63:32] ← 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 *sel* field, and when this is the case, the *sel* field must be set to zero.

When the Coprocessor 0 destination register specified is the *EntryLo0* or *EntryLo1* register, bits 1:0 of the GPR appear at bits 31:30 of *EntryLo0* or *EntryLo1*. This is to compensate for *RI* and *XI*, which were shifted to bits 63:62 by MTC0 to *EntryLo0* or *EntryLo1*. If *RI*/*XI* are not supported, the shift must still occur, but an MFHC0 instruction will return 0s for these two fields. The GPR is right-shifted by two to vacate the lower two bits, and two 0s are shifted in from the left. The result is written to the upper 32 bits of MIPS64 *EntryLo0* or *EntryLo1*, excluding *RI*/*XI*. which were placed in bits 63:62, i.e., the write must appear atomic, as if both MTC0 and MTHC0 occurred together.

This feature supports MIPS32 backward compatibility of MIPS64 systems.

Restrictions:

The results are **UNDEFINED** if Coprocessor 0 does not contain a register as specified by *rd* and *sel*, or if the register exists but is not extended by 32 bits, or the register is extended for XPA, but XPA is not supported or enabled.

In a 64-bit processor, the MTHC0 instruction writes only the lower 32 bits of register *rt* into the upper 32 bits of the Coprocessor register specified by *rd* and *sel* if that register is extended by MIPS32 Release 5. Specifically, the only registers extended by MIPS32 Release 5 are those required for the XPA feature, and those registers are identical to the same registers in the MIPS64 Architecture, other than *EntryLo0* and *EntryLo1*.

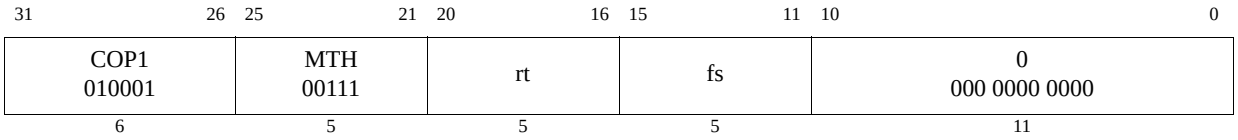
Operation:

```
data ← GPR[rt]
reg ← rd
if (reg,sel = EntryLo1 or reg,sel = EntryLo0) then
       if (Config3LPA = 1 and PageGrainELPA = 1) then // PABITS > 36
          CPR[0,reg,sel]31..30 ← data1..0
          CPR[0,reg,sel]61:32 ← data31..2 and ((1<<(PABITS-36))-1)
          CPR[0,reg,sel]63:62 ← 02
       endif
   else
   CPR[0,reg,sel][63:32] ← data31..0
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction



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

Purpose: Move Word to High Half of Floating Point Register

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)
```

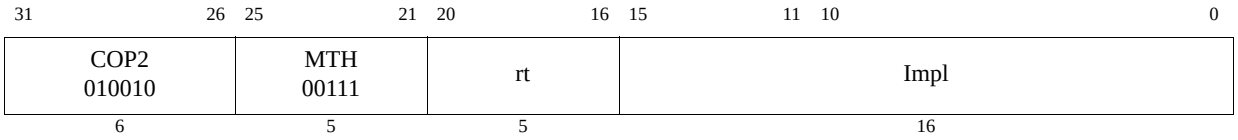
Exceptions:

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.



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

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

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

Purpose: Move Word to High Half of Coprocessor 2 Register

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

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

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
```

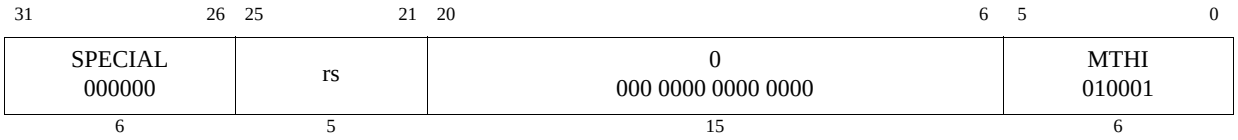
Exceptions:

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.



Format: `MTHI rs` **MIPS32,**

Purpose: Move to HI Register

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

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:

```
MULT   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
```

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

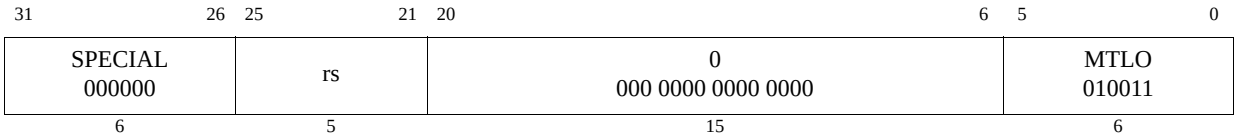
```
HI ← GPR[rs]
```

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.



Format: `MTLO rs` **MIPS32,**

Purpose: Move to LO Register

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:

This instruction has been removed in the MIPS32 Release 6 architecture.

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:

```
MULT   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
```

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

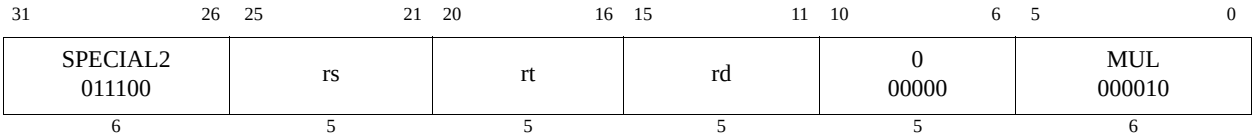
```
LO ← GPR[rs]
```

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.



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

Purpose: Multiply Word to GPR

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.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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.

This pre-MIPS32 Release 6 instruction, MUL, has been removed[1] by MIPS32 Release 6 and replaced by a similar instruction of the same mnemonic, MUL, but different encoding, which is a member of a family of single-width multiply instructions. Refer to the ‘MUL’ and ‘MUH’ instructions in this manual for more information.

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	MUL 00010	011000
SPECIAL 000000	rs	rt	rd	MUH 00011	011000
SPECIAL 000000	rs	rt	rd	MULU 00010	011001
SPECIAL 000000	rs	rt	rd	MUHU 00011	011001
SPECIAL 000000	rs	rt	rd	DMUL 00010	111000
SPECIAL 000000	rs	rt	rd	DMUH 00011	011100
SPECIAL 000000	rs	rt	rd	DMULU 00010	111001
SPECIAL 000000	rs	rt	rd	DMUHU 00011	111001
6	5	5	5	5	6

Format: `MUL MUH MULU MUHU DMUL DMUH DMULU DMUHU`

```
MUL rd,rs,rt                    MIPS32 Release 6
MUH rd,rs,rt                    MIPS32 Release 6
MULU rd,rs,rt                   MIPS32 Release 6
MUHU rd,rs,rt                   MIPS32 Release 6
DMUL rd,rs,rt                   MIPS64 Release 6
DMUH rd,rs,rt                   MIPS64 Release 6
DMULU rd,rs,rt                  MIPS64 Release 6
DMUHU rd,rs,rt                  MIPS64 Release 6
```

Preliminary

Purpose: Multiply Integers (with result to GPR)

MUL: Multiply Words Signed, Low Word
MUH: Multiply Words Signed, High Word
MULU: Multiply Words Signed, Low Word
MUHU: Multiply Words Signed, High Word
DMUL: Multiply Doublewords Signed, Low Doubleword
DMUH: Multiply Doublewords Signed, High Doubleword
DMULU: Multiply Doublewords Signed, Low Doubleword
DMUHU: Multiply Doublewords Signed, High Doubleword

Description:

```
MUL:   GPR[rd] ← sign_extend.32( lo_word( multiply.signed( GPR[rs] × GPR[rt] )
MUH:   GPR[rd] ← sign_extend.32( hi_word( multiply.signed( GPR[rs] × GPR[rt] )
MULU:  GPR[rd] ← sign_extend.32( lo_word( multiply.unsigned( GPR[rs] ×GPR[rt] )
MUHU:  GPR[rd] ← sign_extend.32( hi_word( multiply.unsigned( GPR[rs] ×GPR[rt] )
DMUL:  GPR[rd] ← lo_doubleword( multiply.signed( GPR[rs] × GPR[rt] )
DMUH:  GPR[rd] ← hi_doubleword( multiply.signed( GPR[rs] × GPR[rt] )
DMULU: GPR[rd] ← lo_doubleword( multiply.unsigned( GPR[rs] × GPR[rt] )
DMUHU: GPR[rd] ← hi_doubleword( multiply.unsigned( GPR[rs] ×GPR[rt] )
```

The MIPS32 Release 6 multiply instructions multiply the operands in GPR[rs] and GPR[rd], and place the specified high or low part of the result, of the same width, in GPR[rd].

MUL performs a signed 32-bit integer multiplication, and places the low 32 bits of the result in the destination register.

MUH performs a signed 32-bit integer multiplication, and places the high 32 bits of the result in the destination register.

MULU performs an unsigned 32-bit integer multiplication, and places the low 32 bits of the result in the destination register.

MUHU performs an unsigned 32-bit integer multiplication, and places the high 32 bits of the result in the destination register.

DMUL performs a signed 64-bit integer multiplication, and places the low 64 bits of the result in the destination register.

DMUH performs a signed 64-bit integer multiplication, and places the high 64 bits of the result in the destination register.

DMULU performs an unsigned 64-bit integer multiplication, and places the low 64 bits of the result in the destination register.

DMUHU performs an unsigned 64-bit integer multiplication, and places the high 64 bits of the result in the destination register.

Restrictions:

On a 64-bit CPU, MUH is UNPREDICTABLE if its inputs are not signed extended 32-bit integers.

MUL behaves correctly even if its inputs are not sign extended 32-bit integers. Bits 32-63 of its inputs do not affect the result.

Special provision is made for the inputs to unsigned 32-bit multiplies on a 64-bit CPU. Since many instructions produce sign extend 32 bits to 64 even for unsigned computation, properly sign extended numbers must be accepted as input, and truncated to 32 bits, clearing bits 32-63. However, it is also desirable to accept zero extended 32-bit integers, with bits 32-63 all 0.[1]

On a 64-bit CPU, MUHU is UNPREDICTABLE if its inputs are not zero or sign extended 32-bit integers.

MULU behaves correctly even if its inputs are not zero or sign extended 32-bit integers. Bits 32-63 of its inputs do not affect the result.

On a 64-bit CPU, the 32-bit multiplications, both signed and unsigned, sign extend the result as if it is a 32-bit signed integer.

Availability:

These instructions are introduced by and required as of Release 6.

Programming Notes:

The low half of the integer multiplication result is identical for signed and unsigned. Nevertheless, there are distinct instructions MUL MULU DMUL DMULU. Implementations may choose to optimize a multiply that produces the low half followed by a multiply that produces the upper half. Programmers are recommended to use matching lower and upper half multiplications.

The MIPS32 Release 6 MUL instruction (000000.rs.rt.rd.00010.011000) has the same opcode mnemonic as the pre-

1. Requiring that both zero or sign extended integers be accepted may complicate the multiplier. TBD: require only 32-bit signed inputs?

MIPS32 Release 6 MUL instruction (000000.rs.rt.00000.000010). The semantics of these instructions are almost identical: both produce the low 32-bits of the 32×32=64 product; but the pre-MIPS32 Release 6 MUL is unpredictable if its inputs are not properly sign extended 32-bit values on a 64 bit machine, and is defined to render the HI and LO registers unpredictable, whereas the MIPSt6 version ignores bits 32-63 of the input, and there are no HI/LO registers in MIPS32 Release 6 to be affected. Little is lost by using the same mnemonic. If disambiguation is necessary, say MIPS32 Release 6 versus pre-MIPS Release 6, or specify the instruction string.

The pre-MIPS32 Release 6 MUL instruction (000000.rs.rt.00000.000010) works on MIPS32 Release 6[2], but is deprecated. New software should not use this instruction.

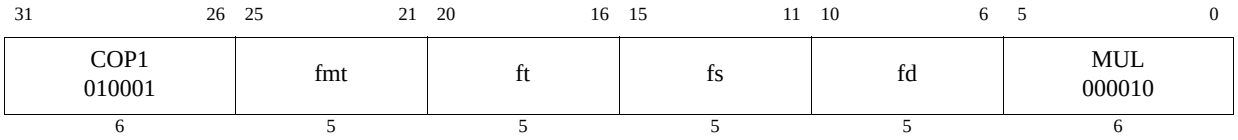
Operation

```
MUH: if NotWordValue(GPR[rs]) then UNPREDICTABLE ?
MUH: if NotWordValue(GPR[rt]) then UNPREDICTABLE ?
MUHU: if not(zero_or_sign_extended.32(GPR[rs])) then UNPREDICTABLE ?
MUHU: if not(zero_or_sign_extended.32(GPR[rt])) then UNPREDICTABLE ?

/* recommended imnplementation: ignore bits 32-63 for MUL, MUH, MULU, MUHU */

MUL, MUH:
  s1 ← signed_word(GPR[rs])
  s2 ← signed_word(GPR[rt])
MULU, MUHU:
  s1 ← unsigned_word(GPR[rs])
  s2 ← unsigned_word(GPR[rt])
DMUL, DMUH:
  s1 ← signed_doubleword(GPR[rs])
  s2 ← signed_dpubleword(GPR[rt])
DMULU, DMUHU:
  s1 ← unsigned_doubleword(GPR[rs])
  s2 ← unsigned_doubleword(GPR[rt])

product ← s1 × s2     /* product is twice the width of sources */

MUL:   GPR[rd] ← sign_extend.32( lo_word( product )
MUH:   GPR[rd] ← sign_extend.32( hi_word( product )
MULU:  GPR[rd] ← sign_extend.32( lo_word( product )
MUHU:  GPR[rd] ← sign_extend.32( hi_word( product )
DMUL:  GPR[rd] ← lo_doubleword( product )
DMUH:  GPR[rd] ← hi_doubleword( product )
DMULU: GPR[rd] ← lo_doubleword( product )
DMUHU: GPR[rd] ← hi_doubleword( product )
?

where

   function zero_or_sign_extended.32(val)
      if value63..32 = (value31)32 then return true
      if value63..32 = (0)32 then return true
      return false
   end function
```

2. TBD: verify that pre-MIPSt6 MUL has not been allocated to some other instruction.

Exceptions:

None



Format: `MUL.fmt`

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

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

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

Purpose: Floating Point Multiply

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

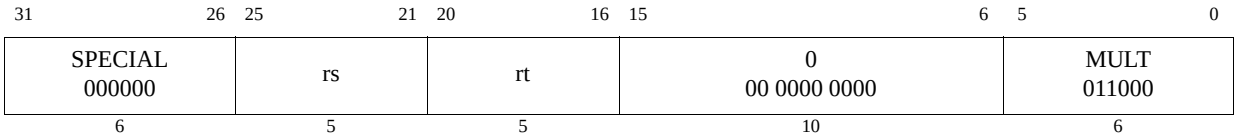
```
StoreFPR (fd, fmt, ValueFPR(fs, fmt) ×fmt ValueFPR(ft, fmt))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow



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

Purpose: Multiply Word

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**.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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:

This instruction, MULT, which places the high and low halves of the product in the HI and LO special registers, has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the Multiply Low (MUL) and Multiply Hi (MUH) instructions, whose output is written to a single GPR. Refer to the ‘MUL’ and ‘MUH’ instructions in this manual for more information.

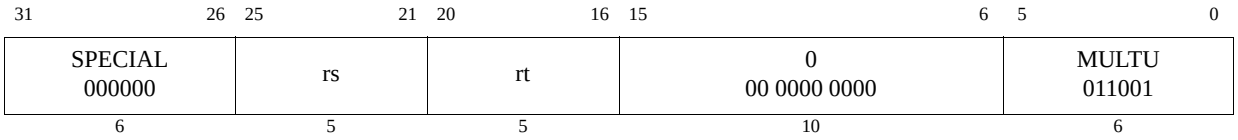
In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result,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

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.

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.

For implementations that do not implement the DSP Module, it is recommended that this instruction should cause a Reserved Instruction Exception if bits 15:11 are not zero. This is to enable emulation of DSP Module instructions.



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

Purpose: Multiply Unsigned Word

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**.

This instruction, MULTU, which places the high and low halves of the product in the HI and LO special registers, has been removed[1] by the MIPS32 Release 6 architecture and has been replaced by the Multiply Low (MULU) and Multiply Hi (MUHU) instructions, whose output is written to a single GPR. Refer to the ‘MULU’ and ‘MUHU’ instructions in this manual for more information.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

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:

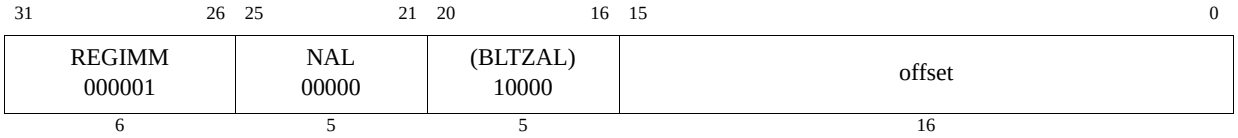
In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result,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

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.

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.

For implementations that do not implement the DSP Module, it is recommended that this instruction should cause a Reserved Instruction Exception if bits 15:11 are not zero. This is to enable emulation of DSP Module instructions.



Format: `NAL`
`NAL offset` **MIPS32 Release 6**

Purpose: No-op and Link

Description: `GPR[31] ← PC+8`

NAL is a deprecated instruction to read the program counter.

NAL was a convenient way to read the PC in the pre-MIPS32 Release 6 instruction set architecture.

NAL was originally an alias for pre-MIPS32 Release 6 instruction `BLTZAL r0,IgnoredTarget` `000001.rs.10000.offset16`. The condition is false, so the 16-bit target offset field is ignored, but the link register, GPR 31, is unconditionally written with the address of the instruction past the delay slot.

MIPS32 Release 6 removes BLTZAL, the instruction of which NAL was a special case. MIPS32 Release 6 implementations are required to signal a Reserved Instruction Exception when BLTZAL is encountered, except as specified below.

MIPS32 Release 6 deprecates NAL, but continues to support this encoding without trapping. NAL may therefore be considered an instruction in its own right in MIPS32 Release 6.

`ADDIUPC` is recommended as a more powerful way of generating a PC-relative address.

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

Restrictions:

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

NAL is considered to be a not-taken branch, with a delay slot, and may not be followed by instructions not allowed in delay slots. Nor is NAL allowed in a delay slot or forbidden slot.

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE. See 4.1.3 “Jump and Branch Instructions” on page 86 in Volume I for detailed discussion of correctness and performance issues.

Prior to MIPS32 Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

MIPS32 Release 6: If a control transfer instruction (CTI) is placed in the delay slot of a branch or jump, MIPS Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:

Prior to MIPS32 Release 6 instruction `BLTZAL, 000001.rs.10000.offset16,` when `rs` is not GPR[0] is removed in MIPS32 Release 6, and is required to signal a Reserved Instruction exception. MIPS32 Release 6 adds `BLTZALC`, the equivalent compact conditional branch and link, with no delay slot.

This instruction, NAL, is introduced by and required as of MIPS32 Release 6, in the sense that the mnemonic NAL becomes officially distinguished from the `BLTZAL` instruction removed by MIPS32 Release 6. The NAL instruction encoding, however, works on all MIPS implementations, both pre-MIPS32 Release 6, where it was a special case of BLEZAL, and MIPS32 Release 6, where it is an instruction in its own right.

The instruction NAL is also deprecated by MIPS32 Release 6, even though NAL is introduced by and required as of MIPS32 Release 6. NAL is provided only for compatibility with pre-MIPS32 Release 6 software. ADDIUPC is recommended as a more powerful way of generating a PC-relative address.

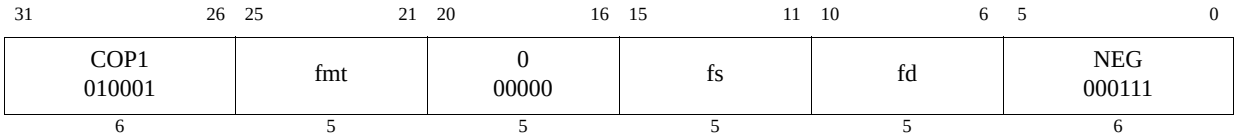
Exceptions:

None[1]

Operation:

```
GPR[31] ← PC + 8
```

1. Except for possible Reserved Instruction Exception if used in a Forbidden Slot, true of these and most other control transfer instructions (CTIs)



Format: `NEG.fmt`

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

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

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

Purpose: Floating Point Negate

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.

If $FIR_{Has2008}$=0 or $FCSR_{ABS2008}$=0 then this operation is arithmetic. For this case, any NaN operand signals invalid operation.

If $FCSR_{ABS2008}$=1 then this operation is non-arithmetic. For this case, both regular floating point numbers and NAN values are treated alike, only the sign bit is affected by this instruction. No IEEE exception can be generated for this case.

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

This NEG.PS instruction has been removed in the Release 6 architecture.

Operation:

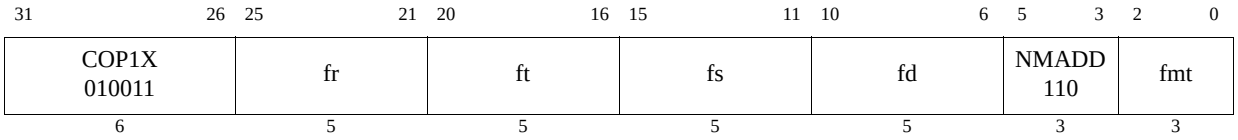
```
StoreFPR(fd, fmt, Negate(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation



Format: `NMADD.fmt`

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

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

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

Purpose: Floating Point Negative Multiply Add

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 intermediate product is rounded according to the current rounding mode in *FCSR*. The value in FPR *fr* is added to the product.

The result sum is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, negated by changing the sign bit, and placed into FPR *fd*. The operands and result are values in format *fmt*. The results and flags are as if separate floating-point multiply and add and negate instructions were executed.

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 the *FR*=0 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Compatibility and Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

NMADD.S and NMADD.D: Required in all versions of MIPS64 since MIPS64 Release 1. Not available in MIPS32 Release 1. Required by MIPS32 Release 2 and subsequent versions of MIPS32.

Required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (FIR_{F64}=0 or 1, $Status_{FR}$=0 or 1).

Operation:

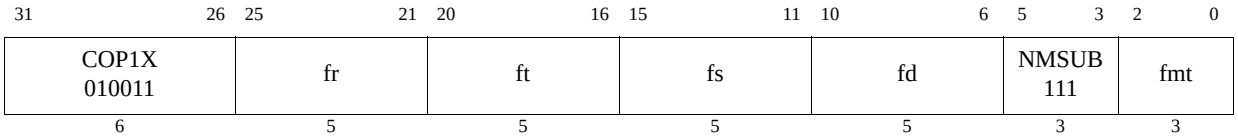
```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, –(vfr +fmt (vfs ×fmt vft)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow



Format: `NMSUB.fmt`

`NMSUB.S fd, fr, fs, ft` **MIPS64, MIPS32 Release 2**

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

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

Purpose: Floating Point Negative Multiply Subtract

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 intermediate product is rounded according to the current rounding mode in *FCSR*. The value in FPR *fr* is subtracted from the product.

The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, negated by changing the sign bit, and placed into FPR *fd*. The operands and result are values in format *fmt*. The results and flags are as if separate floating-point multiply and subtract and negate instructions were executed.

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 the *FR*=0 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Compatibility and Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

NMSUB.S and NMSUB.D: Required in all versions of MIPS64 since MIPS64 Release 1. Not available in MIPS32 Release 1. Required by MIPS32 Release 2 and subsequent versions of MIPS32.

Required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (FIR_{F64}=0 or 1, $Status_{FR}$=0 or 1).

Operation:

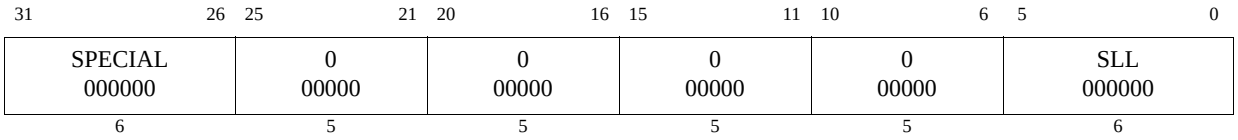
```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, –((vfs ×fmt vft) –fmt vfr))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow



Format: `NOP` **Assembly Idiom**

Purpose: No Operation

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

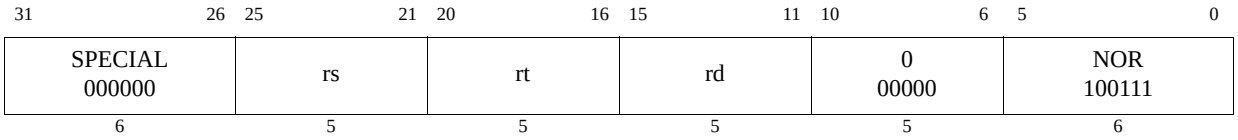
Exceptions:

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.

NOP	No Operation



Format: `NOR rd, rs, rt` **MIPS32**

Purpose: Not Or

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:

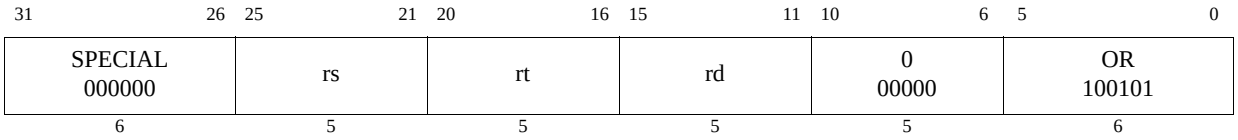
None

Operation:

```
GPR[rd] ← GPR[rs] nor GPR[rt]
```

Exceptions:

None



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

Purpose: Or

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:

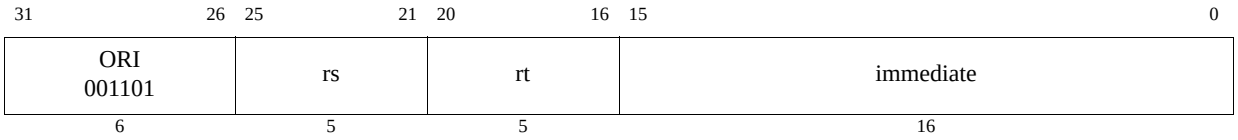
None

Operation:

```
GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:

None



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

Purpose: Or Immediate

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:

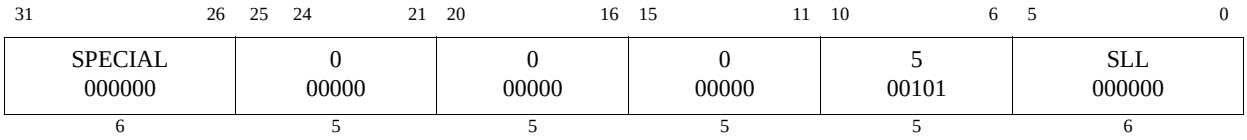
None

Operation:

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

Exceptions:

None



Format: `PAUSE` **MIPS32 Release 2/MT Module**

Purpose: Wait for the LLBit to clear

Description:

Locks implemented using the LL/SC (or LLD/SCD) instructions are a common method of synchronization between threads of control. A typical lock implementation does a load-linked instruction and checks the value returned to determine whether the software lock is set. If it is, the code branches back to retry the load-linked instruction, thereby implementing an active busy-wait sequence. The PAUSE instructions is intended to be placed into the busy-wait sequence to block the instruction stream until such time as the load-linked instruction has a chance to succeed in obtaining the software lock.

The precise behavior of the PAUSE instruction is implementation-dependent, but it usually involves descheduling the instruction stream until the LLBit is zero. In a single-threaded processor, this may be implemented as a short-term WAIT operation which resumes at the next instruction when the LLBit is zero or on some other external event such as an interrupt. On a multi-threaded processor, this may be implemented as a short term YIELD operation which resumes at the next instruction when the LLBit is zero. In either case, it is assumed that the instruction stream which gives up the software lock does so via a write to the lock variable, which causes the processor to clear the LLBit as seen by this thread of execution.

The encoding of the instruction is such that it is backward compatible with all previous implementations of the architecture. The PAUSE instruction can therefore be placed into existing lock sequences and treated as a NOP by the processor, even if the processor does not implement the PAUSE instruction.

Restrictions:

Prior to MIPS32 Release 6: The operation of the processor is **UNDEFINED** if a PAUSE is executed in the delay slot of a branch or jump instruction.

MIPS32 Release 6 implementations are required to signal a Reserved Instruction Exception if PAUSE is encountered in the delay slot or forbidden slot of a branch or jump instruction.

Operation:

```
if LLBit ≠ 0 then
   EPC ← PC + 4                  /* Resume at the following instruction */
   DescheduleInstructionStream()
endif
```

Exceptions:

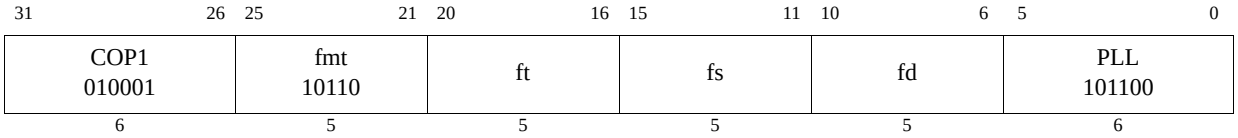
None

Programming Notes:

The PAUSE instruction is intended to be inserted into the instruction stream after an LL instruction has set the LLBit and found the software lock set. The program may wait forever if a PAUSE instruction is executed and there is no possibility that the LLBit will ever be cleared.

An example use of the PAUSE instruction is included in the following example:

```
acquire_lock:
   ll     t0, 0(a0)              /* Read software lock, set hardware lock */
   bnez   t0, acquire_lock_retry: /* Branch if software lock is taken */
   addiu  t0, t0, 1              /* Set the software lock */
   sc     t0, 0(a0)              /* Try to store the software lock */
   bnez   t0, 10f                /* Branch if lock acquired successfully */
   sync
acquire_lock_retry:
   pause                         /* Wait for LLBIT to clear before retry */
   b      acquire_lock           /* and retry the operation */
   nop
10:

   Critical region code

release_lock:
   sync
   sw     zero, 0(a0)            /* Release software lock, clearing LLBIT */
                                 /* for any PAUSEd waiters */
```



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

Purpose: Pair Lower Lower

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 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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

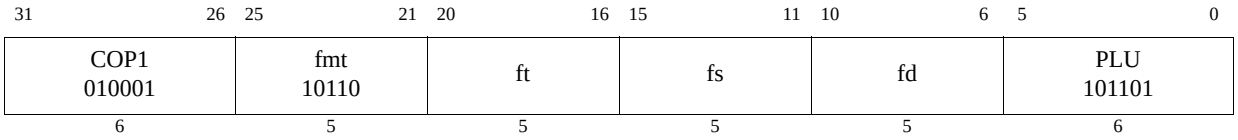
This instruction has been removed in the Release 6 architecture.

Operation:

```
StoreFPR(fd, PS, ValueFPR(fs, PS)31..0 || ValueFPR(ft, PS)31..0)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction



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

Purpose: Pair Lower Upper

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

This instruction has been removed in the Release 6 architecture.

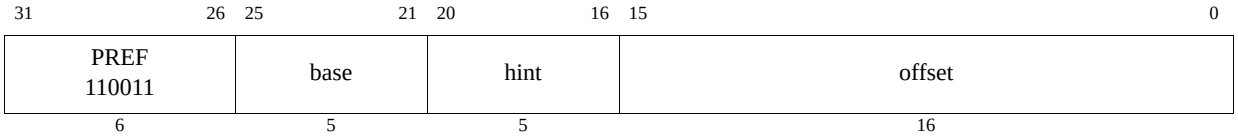
Operation:

```
StoreFPR(fd, PS, ValueFPR(fs, PS)31..0 || ValueFPR(ft, PS)63..32)
```

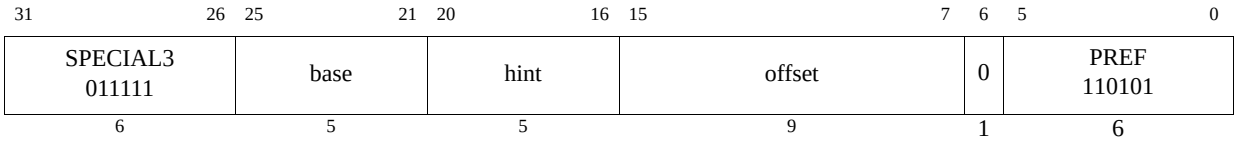
Exceptions:

Coprocessor Unusable, Reserved Instruction

MIPS32, (all release levels less than Release 6)



MIPS32 Release 6 (Release 6 and future)



Format: `PREF hint,offset(base)` **MIPS32**

Purpose: Prefetch

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 cacheability and coherency attribute of a segment (e.g., the use of the *K0*, *KU*, or *K23* fields in the *Config* register), or the per-page cacheability and coherency attribute provided by the TLB.

If PREF results in a memory operation, the memory access type and cacheability&coherency attribute used for the operation are determined by the memory access type and cacheability&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.

In coherent multiprocessor implementations, if the effective address uses a coherent Cacheability and Coherency Attribute (CCA), then the instruction causes a coherent memory transaction to occur. This means a prefetch issued on one processor can cause data to be evicted from the cache in another processor.

The PREF instruction and the memory transactions which are sourced by the PREF instruction, such as cache refill or cache writeback, obey the ordering and completion rules of the SYNC instruction.

Table 4.13 Values of *hint* Field for 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	Reserved Implementation Dependent	Reserved for future use - not available to implementations. Implementation dependent in MIPS32 Release 6. This hint code marks the line as LRU in the L1 cache and thus preferred for next eviction. Implementations can choose to writeback and/or invalidate as long as no architectural state is modified.
3	Reserved Implementation Dependent	Reserved for future use - not available to implementations. This hint code is unused in the MIPS32 Release 6 architecture.
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.”
8-15	Reserved	In the MIPS32 Release 6 architecture, hint codes 8 - 15 are treated the same as hint codes 0 - 7 respectively, but operate on the L2 cache.
16-23	Reserved	In the MIPS32 Release 6 architecture, hint codes 16 - 23 are treated the same as hint codes 0 - 7 respectively, but operate on the L3 cache.
8-20	Reserved	Reserved for future use - not available to implementations.
21-23	Implementation Dependent Implementation Dependent	Unassigned by the Architecture - available for implementation-dependent use. Unassigned by the Architecture - available for implementation-dependent use.
24	Implementation Dependent Reserved in MIPr6	Unassigned by the Architecture - available for implementation-dependent use. This hint code is not implemented in the MIPS32 Release 6 architecture and generates a Reserved Instruction exception (RI).

Table 4.13 Values of *hint* Field for PREF Instruction

25	writeback_invalidate (also known as “nudge”) Reserved in MIPS Release 6	Use: Data is no longer expected to be used. Action: For a writeback cache, schedule a writeback 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. This hint code is not implemented in the MIPS32 Release 6 architecture and generates a Reserved Instruction exception (RI).
26-29	Implementation Dependent Reserved in MIPS Release 6	Unassigned by the Architecture - available for implementation-dependent use. These hints are not implemented in the MIPS32 Release 6 architecture and generate a Reserved Instruction exception (RI).
30	PrepareForStore Reserved in MIPS Release 6	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. This hint is not implemented in the MIPS32 Release 6 architecture and generates a Reserved Instruction exception (RI).
31	Implementation Dependent Reserved in MIPr6	Unassigned by the Architecture - available for implementation-dependent use. This hint is not implemented in the MIPS32 Release 6 architecture and generates a Reserved Instruction exception (RI).

Restrictions:

None.

This instruction does not produce an exception for a misaligned memory address, since it has no memory access size.

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:

In the MIPS32 Release 6 architecture, hint codes 0:23 behave as a NOP and never signal a Reserved Instruction exception (RI). Hint codes 24:31 are not implemented (treated as reserved) and always signal a Reserved Instruction exception (RI).

As shown in the instruction drawing above, the MIPS Release 6 architecture implements a 9-bit offset, whereas all release levels lower than Release 6 of the MIPS architecture implement a 16-bit offset.

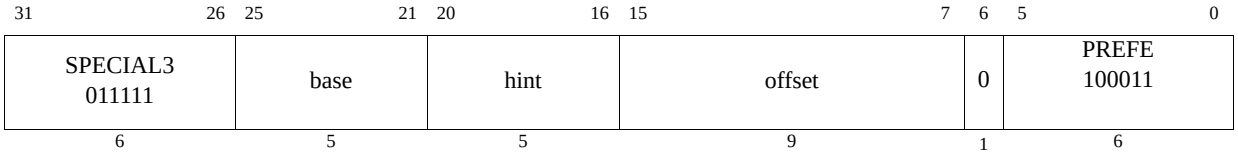
Prefetch cannot move data to or from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

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.



Format: `PREFE hint,offset(base)` **MIPS32**

Purpose: Prefetch EVA

To move data between user mode virtual address space memory and cache while operating in kernel mode.

Description: `prefetch_memory(GPR[base] + offset)`

PREFE adds the 9-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.

PREFE 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 PREFE 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.

PREFE 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 PREFE instruction.

PREFE 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 cacheability and coherency attribute of a segment (e.g., the use of the *K0*, *KU*, or *K23* fields in the *Config* register), or the per-page cacheability and coherency attribute provided by the TLB.

If PREFE results in a memory operation, the memory access type and cacheability&coherency attribute used for the operation are determined by the memory access type and cacheability&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.

In coherent multiprocessor implementations, if the effective address uses a coherent Cacheability and Coherency Attribute (CCA), then the instruction causes a coherent memory transaction to occur. This means a prefetch issued on one processor can cause data to be evicted from the cache in another processor.

The PREFE instruction and the memory transactions which are sourced by the PREFE instruction, such as cache refill or cache writeback, obey the ordering and completion rules of the SYNC instruction.

The PREFE instruction functions in exactly the same fashion as the PREF instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Table 4.14 Values of *hint* Field for PREFE 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	Reserved Implementation Dependent	Reserved for future use - not available to implementations. Implementation dependent in MIPS32 Release 6. This hint code marks the line as LRU in the L1 cache and thus preferred for next eviction. Implementations can choose to writeback and/or invalidate as long as no architectural state is modified.
3	Reserved Implementation Dependent	Reserved for future use - not available to implementations. This hint code is unused in the MIPS32 Release 6 architecture.
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.”
8-15	Reserved	In the MIPS32 Release 6 architecture, hint codes 8 - 15 are treated the same as hint codes 0 - 7 respectively, but operate on the L2 cache.
16-23	Reserved	In the MIPS32 Release 6 architecture, hint codes 16 - 23 are treated the same as hint codes 0 - 7 respectively, but operate on the L3 cache.
8-20	Reserved	Reserved for future use - not available to implementations.
21-23	Implementation Dependent	Unassigned by the Architecture - available for implementation-dependent use.
24	Implementation Dependent Reserved in MIPS Release 6	Unassigned by the Architecture - available for implementation-dependent use. This hint code is not implemented in the MIPS32 Release 6 architecture and generates a Reserved Instruction exception (RI).

Table 4.14 Values of *hint* Field for PREFE Instruction

25	writeback_invalidate (also known as “nudge”) Reserved in MIPr6	Use: Data is no longer expected to be used. Action: For a writeback cache, schedule a writeback 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. This hint code is not implemented in the MIPS32 Release 6 architecture and generates a Reserved Instruction exception (RI).
26-29	Implementation Dependent Reserved in MIPS Release 6	Unassigned by the Architecture - available for implementation-dependent use. These hint codes are not implemented in the MIPS32 Release 6 architecture and generate a Reserved Instruction exception (RI).
30	PrepareForStore Reserved in MIPS Release 6	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. This hint code is not implemented in the MIPS32 Release 6 architecture and generates a Reserved Instruction exception (RI).
31	Implementation Dependent Reserved in MIPS Release 6	Unassigned by the Architecture - available for implementation-dependent use. This hint code is not implemented in the MIPS32 Release 6 architecture and generates a Reserved Instruction exception (RI).

Restrictions:

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

This instruction does not produce an exception for a misaligned memory address, since it has no memory access size.

Operation:

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

Exceptions:

Bus Error, Cache Error, Address Error, Reserved Instruction, Coprocessor Usable

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:

In the MIPS32 Release 6 architecture, hint codes 0:23 behave as a NOP and never signal a Reserved Instruction exception (RI). Hint codes 24:31 are not implemented (treated as reserved) and always signal a Reserved Instruction exception (RI).

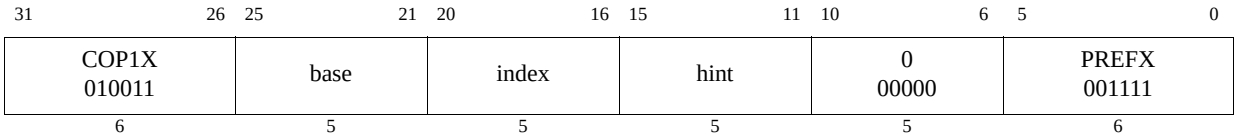
Prefetch cannot move data to or from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

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 PREFE 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.



Format: `PREFX hint, index(base)` **MIPS64, MIPS32 Release 2**

Purpose: Prefetch Indexed

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:

This instruction has been removed[1] in the MIPS32 Release 6 architecture.

Prior to MIPS32 Release 6, the following restrictions apply:

Compatibility and Availability:

PREFX: Required in all versions of MIPS64 since MIPS64 Release 1. Not available in MIPS32 Release 1. Required by MIPS32 Release 2 and subsequent versions of MIPS32.

Required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (FIR_{F64}=0 or 1, $Status_{FR}$=0 or 1).

Operation:

```
vAddr ← GPR[base] + GPR[index]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

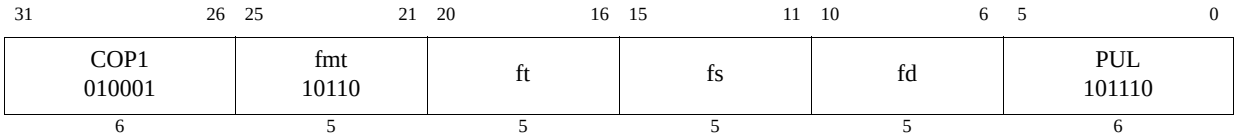
Coprocessor Unusable, Reserved Instruction, Bus Error, Cache Error

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.

Refer to the corresponding section in the PREF instruction description.

1. “Removed by MIPS32 Release 6” means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Pair Upper Lower

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

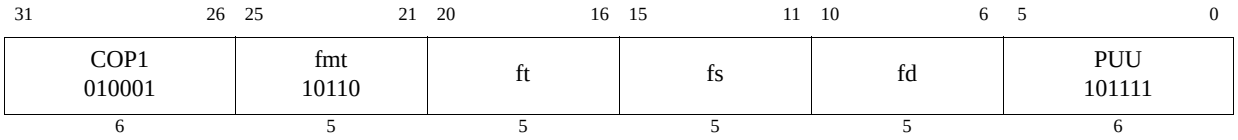
This instruction has been removed in the Release 6 architecture.

Operation:

$$\texttt{StoreFPR(fd, PS, ValueFPR(fs, PS)}_{63..32}\ \texttt{||}\ \texttt{ValueFPR(ft, PS)}_{31..0}\texttt{)}$$

Exceptions:

Coprocessor Unusable, Reserved Instruction



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

Purpose: Pair Upper Upper

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

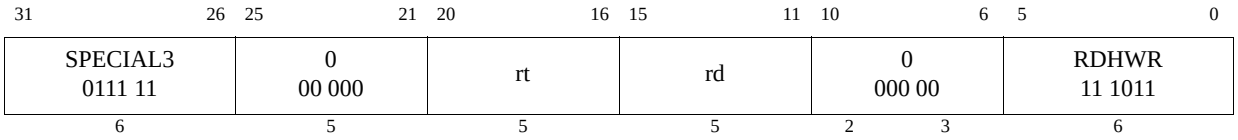
This instruction has been removed in the Release 6 architecture.

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: Read Hardware Register

To move the contents of a hardware register to a general purpose register (GPR) if that operation is enabled by privileged software.

The purpose of this instruction is to give user mode access to specific information that is otherwise only visible in kernel mode.

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 4.15.

Table 4.15 RDHWR Register Numbers

Register Number (*rd* Value)	Mnemonic	Description
0	CPUNum	Number of the CPU on which the program is currently running. This register provides read access to the coprocessor 0 $EBase_{CPUNum}$ field.
1	SYNCI_Step	Address step size to be used with the SYNCI instruction, or zero if no caches need be synchronized. See that instruction's description for the use of this value.
2	CC	High-resolution cycle counter. This register provides read access to 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		These registers numbers are reserved for future architecture use. Access results in a Reserved Instruction Exception.
29	ULR	User Local Register. This register provides read access to the coprocessor 0 *UserLocal* register, if it is implemented. In some operating environments, the *UserLocal* register is a pointer to a thread-specific storage block.

Table 4.15 RDHWR Register Numbers

Register Number (*rd* Value)	Mnemonic	Description
30-31		These register numbers are reserved for implementation-dependent use. 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 or the register is not implemented, a Reserved Instruction Exception is signaled.

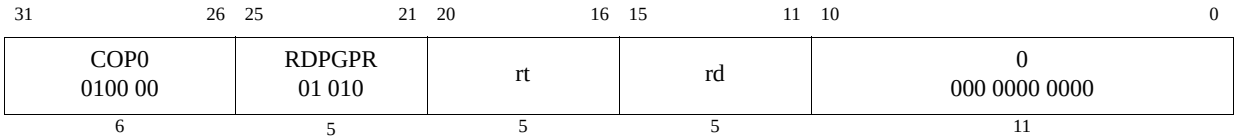
Operation:

```
dcase rd
       0: temp ← sign_extend(EBaseCPUNum)
   1: temp ← sign_extend(SYNCI_StepSize())
   2: temp ← sign_extend(Count)
   3: temp ← sign_extend(CountResolution())
   29: temp ← sign_extend_if_32bit_op(UserLocal)
   30: temp ← sign_extend_if_32bit_op(Implementation-Dependent-Value)
   31: 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

d



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

Purpose: Read GPR from Previous Shadow Set

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.

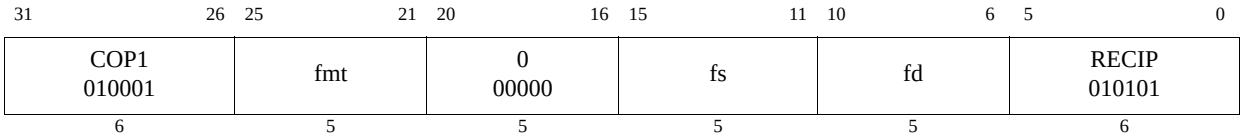
Operation:

```
GPR[rd] ← SGPR[SRSCtl_PSS, rt]
```

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: `RECIP.fmt`

`RECIP.S fd, fs` **MIPS64, MIPS32 Release 2**

`RECIP.D fd, fs` **MIPS64, MIPS32 Release 2**

Purpose: Reciprocal Approximation

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**.

Compatibility and Availability:

RECIP.S and RECIP.D: Required in all versions of MIPS64 since MIPS64 Release1. Not available in MIPS32 Release 1. Required by MIPS32 Release 2 and subsequent versions of MIPS32.

Required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (FIR_{F64}=0 or 1, $Status_{FR}$=0 or 1).

Operation:

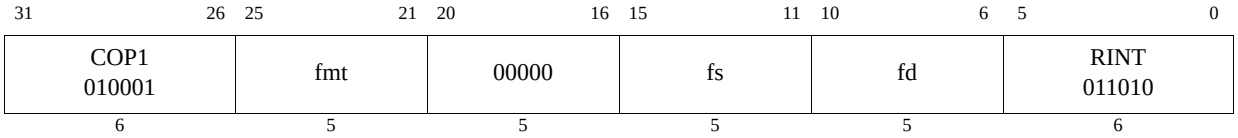
```
StoreFPR(fd, fmt, 1.0 / valueFPR(fs, fmt))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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



Format: `RINT.fmt` **MIPS32 Release 6**
`RINT.S fd,fs` **MIPS32 Release 6**
`RINT.D fd,fs` **MIPS32 Release 6**

Purpose: Floating-Point Round to Integer

Scalar floating-point round to integer.

Description: `FPT[fd] ← round_int(FPR[fs])`

The scalar floating-point value in the register `fs` is rounded to an integral valued floating-point number in the same format based on the rounding mode bits RM in the FPU Control and Status Register *FCSR*. The result is written to `fd`.

The operands and results are values in floating-point data format *fmt*.

The RINT.fmt instruction corresponds to the **roundToIntegralExact** operation in the IEEE Standard for Floating-Point Arithmetic 754™-2008. The Inexact exception is signaled if the result does not have the same numerical value as the input operand.

The floating point scalar instruction RINT.fmt corresponds to the MSA vector instruction FRINT.df. I.e. RINT.S corresponds to FRINT.W, and RINT.D corresponds to FRINT.D.

Restrictions:

Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

Availability:

This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

RINT.fmt:

```
ValidateAccessToFPUResources(fmt,{S,D})

fin ← ValueFPR(fs,fmt)
ftmp ←RoundIntFP(fin, fmt)
if( fin ≠ ftmp ) SignalFPException(InExact)
StoreFPR (fd, fmt, ftmp )

function RoundIntFP(tt, n)
   /* Round to integer operation, using rounding mode FCSR.RM*/
endfunction RoundIntFP

function ValidateAccessToFPUResources(fmt, set_of_permitted_formats)
   /* this function does not return, instruction terminates,
      if exception is signalled, as indicated by ?, without endif
   */
   if not IsCoprocessorEnabled(1) then SignalException(CoprocessorUnusable, 1) ?
   if not IsFloatingPointImplemented(fmt))
      then SignalException(ReservedInstruction)?
```

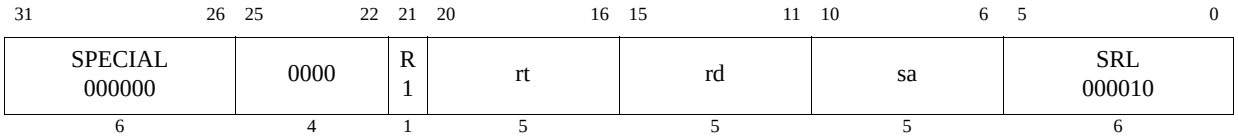
```
        if not ( fmt ∈ set_of_permitted_formats)
           then SignalException(ReservedInstruction)?
        if fmt=S and FIR.S=0 then SignalFPException(UnimplementedOperation)?
        if fmt=D and FIR.D=0 then SignalFPException(UnimplementedOperation)?
        if fmt=W and FIR.W=0 then SignalFPException(UnimplementedOperation)?
        if fmt=L and FIR.L=0 then SignalFPException(UnimplementedOperation)?
        if fmt ∈ {D,L} and not Are64BitFPOperationsEnabled()then
           then SignalException(UnimplementedOperation)
        /* PS (paired single) is deprecated, and is treated separately */
    endfunction ValidateAccessToFPUResources(fmt, set_of_permitted_formats)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow



Format: `ROTR rd, rt, sa` **SmartMIPS Crypto, MIPS32 Release 2**

Purpose: Rotate Word Right

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:

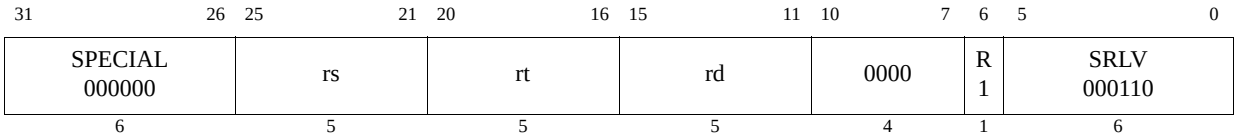
If GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

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



Format: `ROTRV rd, rt, rs` **SmartMIPS Crypto, MIPS32 Release 2**

Purpose: Rotate Word Right Variable

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:

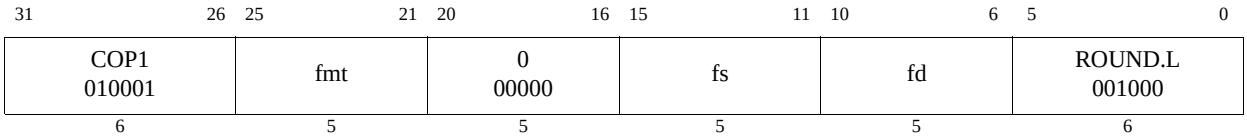
If GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

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.fmt`

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

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

Purpose: Floating Point Round to Long Fixed Point

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

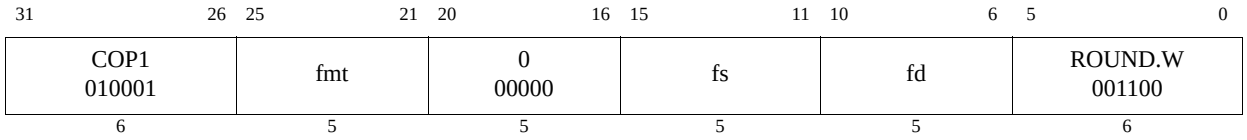
```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation



Format: `ROUND.W.fmt`

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

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

Purpose: Floating Point Round to Word Fixed Point

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:

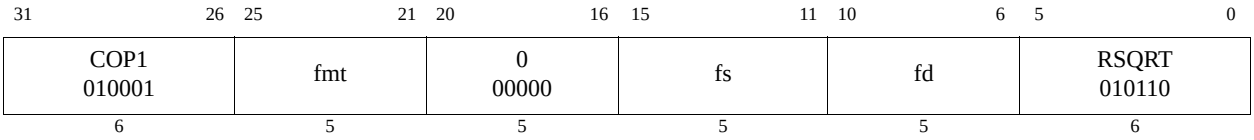
```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation



Format: `RSQRT.fmt`

`RSQRT.S fd, fs` **MIPS64, MIPS32 Release 2**

`RSQRT.D fd, fs` **MIPS64, MIPS32 Release 2**

Purpose: Reciprocal Square Root Approximation

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**.

Compatibility and Availability:

RSQRT.S and RSQRT.D: Required in all versions of MIPS64 since MIPS64 Release 1. Not available in MIPS32 Release 1. Required by MIPS32 Release 2 and subsequent versions of MIPS32.

Required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (FIR_{F64}=0 or 1, $Status_{FR}$=0 or 1).

Operation:

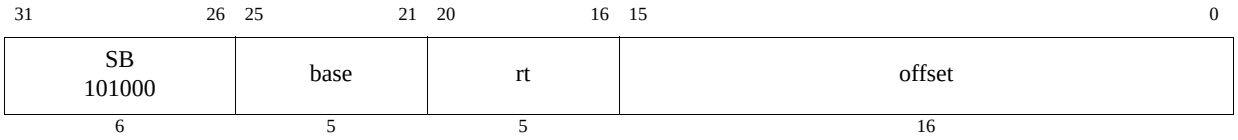
```
StoreFPR(fd, fmt, 1.0 / SquareRoot(valueFPR(fs, fmt)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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



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

Purpose: Store Byte

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:

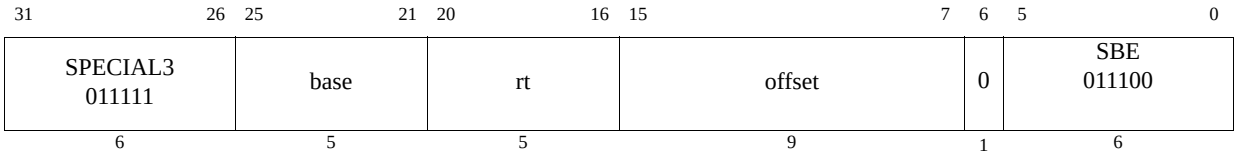
None

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



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

Purpose: Store Byte EVA

To store a byte to user mode virtual address space when executing in kernel mode.

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

The SBE instruction functions in exactly the same fashion as the SB instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Restrictions:

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

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

Bus Error

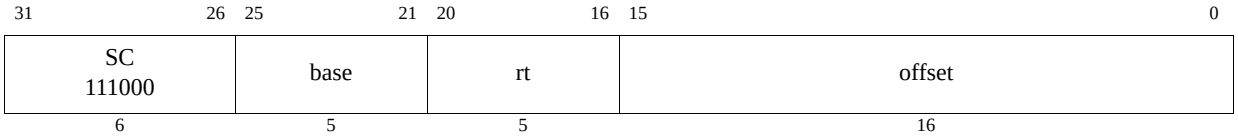
Address Error

Watch

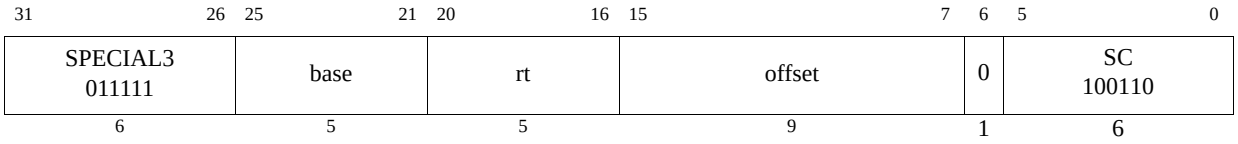
Reserved Instruction

Coprocessor Unusable

MIPS32, (all release levels less than Release 6)



MIPS32 Release 6 (Release 6 and future)



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

Purpose: Store Conditional Word

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 on synchronizable memory locations. In Release 5, the behaviour of SC is modified when $Config5_{LLB}$=1.

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 to memory at the location specified by the aligned effective address.
- A one, 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.
- A coherent store is executed between an LL and SC sequence on the same processor to the block of synchronizable physical memory containing the word (if $Config5_{LLB}$=1; else whether such a store causes the SC to fail is not predictable).
- An ERET instruction is executed. (Release 5 includes ERETNC, which will not cause the SC to fail.)

Furthermore, an SC must always compare its address against that of the LL. An SC will fail if the aligned address of the SC does not match that of the preceeding LL.

A load that executes on the processor executing the LL/SC sequence to the block of synchronizable physical memory containing the word, will not cause the SC to fail (if $Config5_{LLB}$=1; else such a load may cause the SC to fail).

If any of the events listed below occurs between the execution of LL and SC, the SC may fail where it could have succeeded, i.e., success is not predictable. Portable programs should not cause any of these events.

- A load or store executed on the processor executing the LL and SC that is not to the block of synchronizable physical memory containing the word. (The load or store may cause a cache eviction between the LL and SC that results in SC failure. The load or store does not necessarily have to occur between the LL and SC.)
- Any prefetch that is executed on the processor executing the LL and SC sequence (due to a cache eviction between the LL and SC).
- A non-coherent store executed between an LL and SC sequence to the block of synchronizable physical memory containing the word.
- 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.)

CACHE operations that are local to the processor executing the LL/SC sequence will result in unpredictable behaviour of the SC if executed between the LL and SC, that is, they may cause the SC to fail where it could have succeeded. Non-local CACHE operations (address-type with coherent CCA) may cause an SC to fail on either the local processor or on the remote processor in multiprocessor or multi-threaded systems. This definition of the effects of CACHE operations is mandated if $Config5_{LLB}$=1. If $Config5_{LLB}$=0, then CACHE effects are implementation-dependent.

The following conditions must be true or the result of the SC is not predictable—the SC may fail or succeed (if $Config5_{LLB}$=1, then either success or failure is mandated, else the result 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 cacheability & coherency attribute 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.

MIPS Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (AddressError) on a misaligned access is required.

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
LLbit ← 0 // if Config5LLB=1, SC always clears LLbit regardless of address match.
```

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.

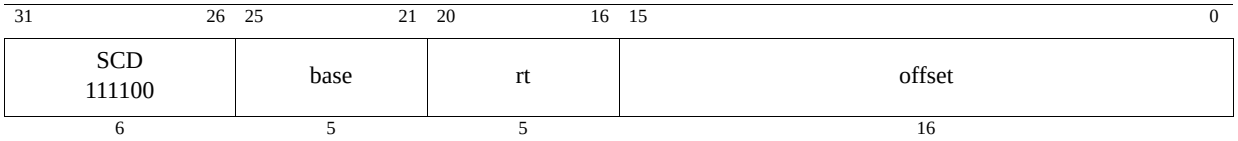
```
L1:
   LL     T1, (T0)  # load counter
   ADDI   T2, T1, 1 # increment
   SC     T2, (T0)  # try to store, checking for atomicity
   BEQ    T2, 0, L1 # if not atomic (0), try again
   NOP              # branch-delay slot
```

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.

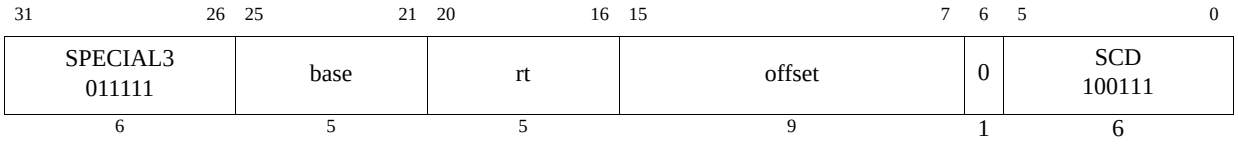
LL and SC function on a single processor for *cached noncoherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.

As shown in the instruction drawing above, the MIPS Release 6 architecture implements a 9-bit offset, whereas all release levels lower than Release 6 of the MIPS architecture implement a 16-bit offset.

MIPS64, (all release levels less than Release 6)



MIPS64 (Release 6 and future)



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

Purpose: Store Conditional Doubleword

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.

MIPS Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (AddressError) on a misaligned access is required.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
datadoubleword ← GPR[rt]
if LLbit then
   StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 063 || LLbit
```

Exceptions:

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

Programming Notes:

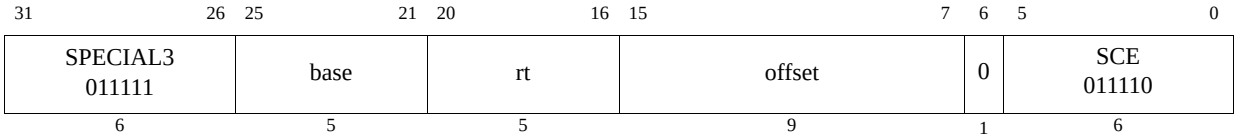
LLD and SCD are used to atomically update memory locations, as shown below.

```
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.



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

Purpose: Store Conditional Word EVA

To store a word to user mode virtual memory while operating in kernel mode 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 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The SCE completes the RMW sequence begun by the preceding LLE 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 to 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 LLE and SCE, the SCE 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 LLE/SCE.
- The instructions executed starting with the LLE and ending with the SCE 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 SCE is **UNPREDICTABLE**:

- Execution of SCE must have been preceded by execution of an LLE instruction.
- An RMW sequence executed without intervening events that would cause the SCE to fail must use the same address in the LLE and SCE. The address is the same if the virtual address, physical address, and cacheability & coherency attribute 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 LLE/SCE 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.

The SCE instruction functions in exactly the same fashion as the SC instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

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.

MIPS Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (AddressError) on a misaligned access is required.

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, Reserved Instruction, Coprocessor Unusable

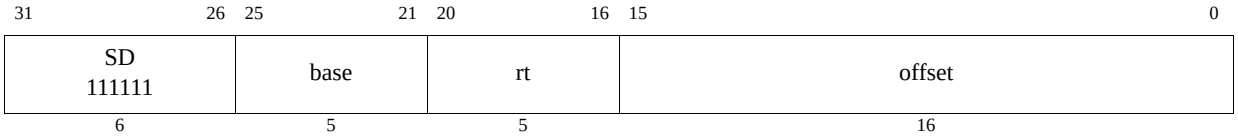
Programming Notes:

LLE and SCE are used to atomically update memory locations, as shown below.

```
L1:
    LLE    T1, (T0)  # load counter
    ADDI   T2, T1, 1 # increment
    SCE    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 LLE and SCE cause SCE 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.

LLE and SCE 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.



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

Purpose: Store Doubleword

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:

Prior to MIPS32 Release 6: 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.

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

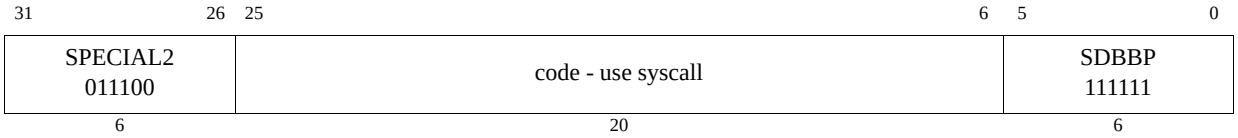
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
       if not MisalignedSupport()
          then SignalException(AddressError)
          else /* implementation dependent misalignment handling */
       endif
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
datadoubleword ← GPR[rt]
StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

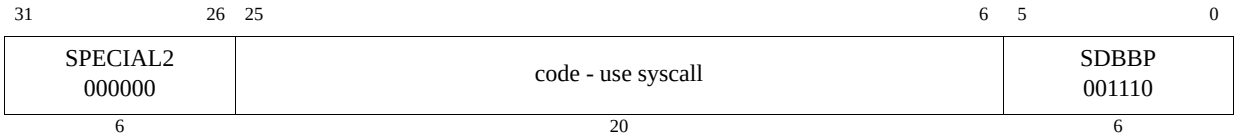
Exceptions:

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

MIPS32, (all release levels less than Release 6)



MIPS32 Release 6 (Release 6 and future)



Format: `SDBBP code` **EJTAG**

Purpose: Software Debug Breakpoint

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 executed, the 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:

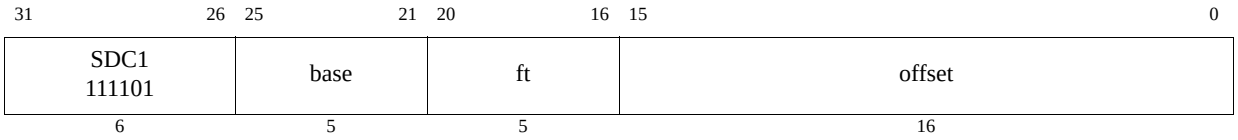
```
if Config5.SBRI=1 then /* SBRI is a MIPS32 Release 6 feature */
   SignalException(ReservedInstruction) endif
If Debug_DM = 1 then SignalDebugModeBreakpointException() endif
SignalDebugBreakpointException()
```

Exceptions:

Debug Breakpoint Exception
Debug Mode Breakpoint Exception

Programming Notes:

As shown in the instruction drawing above, the MIPS Release 6 architecture sets the ‘SPECIAL2’ and ‘SDBBP’ fields to different values than previous releases of the MIPS architecture.



Format: `SDC1 ft, offset(base)` **MIPS32**

Purpose: Store Doubleword from Floating Point

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:

Prior to MIPS32 Release 6: An Address Error exception occurs if $EffectiveAddress_{2..0} \neq 0$ (not doubleword-aligned).

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

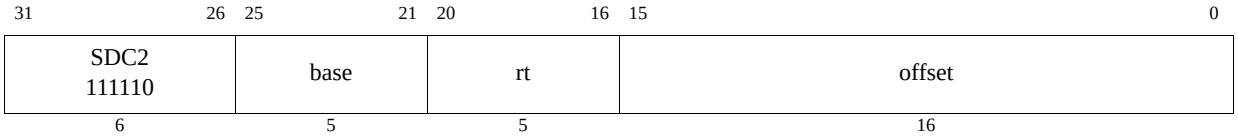
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
       if vAddr2..0 ≠ 03 then
           if not MisalignedSupport()
               then SignalException(AddressError)
               else /* implementation dependent misalignment handling */
           endif
       endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

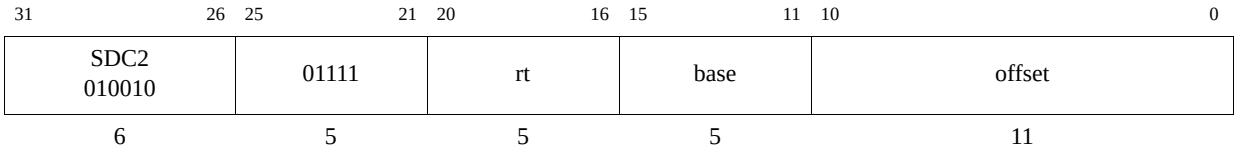
Exceptions:

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

MIPS32, (all release levels less than Release 6)



MIPS32 Release 6 (Release 6 and future)



Format: `SDC2 rt, offset(base)` **MIPS32**

Purpose: Store Doubleword from Coprocessor 2

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:

Prior to MIPS32 Release 6: An Address Error exception occurs if $EffectiveAddress_{2..0} \neq 0$ (not doubleword-aligned).

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

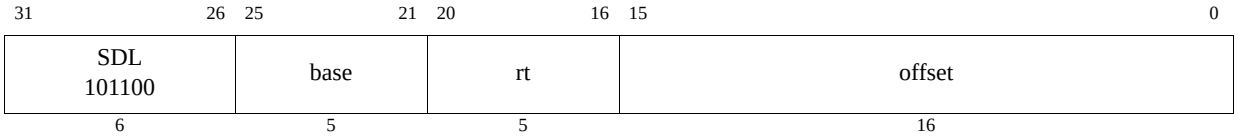
```
vAddr ← sign_extend(offset) + GPR[base]
       if vAddr2..0 ≠ 03 then
           if not MisalignedSupport()
               then SignalException(AddressError)
               else /* implementation dependent misalignment handling */
           endif
       endif
(pAddr, CCA) ← AddressTranslation(vAddr, 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

Programming Notes:

As shown in the instruction drawing above, the MIPS Release 6 architecture implements an 11-bit offset, whereas all release levels lower than Release 6 of the MIPS architecture implement a 16-bit offset.



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

Purpose: Store Doubleword Left

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

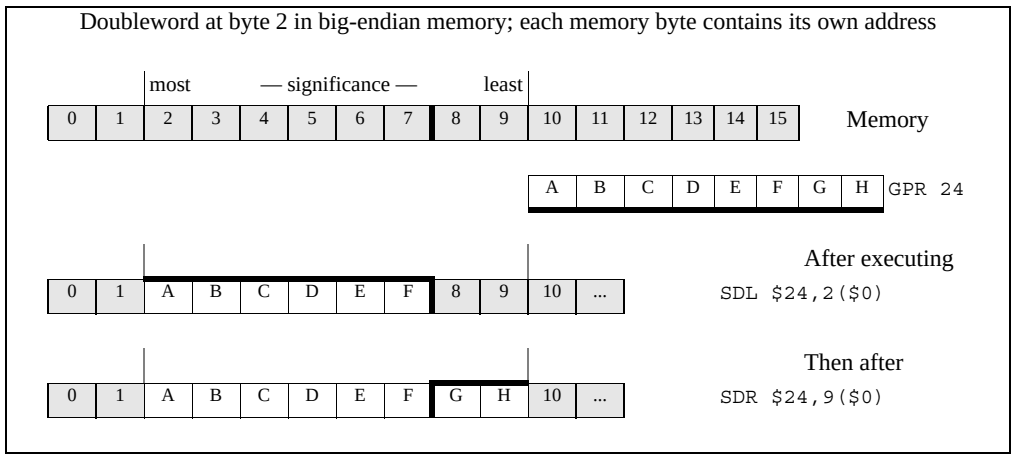
Description: `memory[GPR[base] + offset] ← Some_Bytes_From GPR[rt]`

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*.

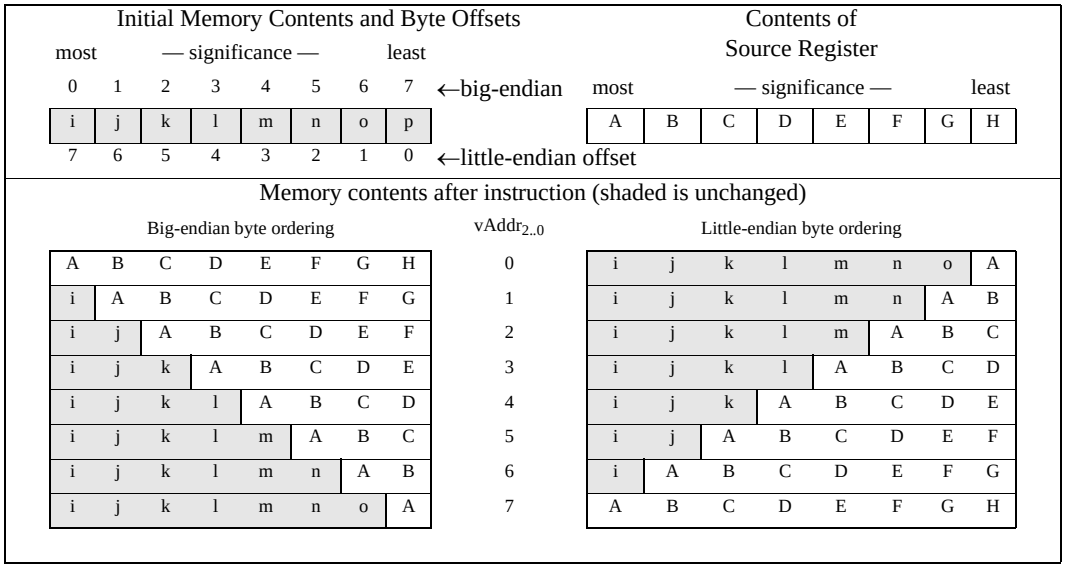
The figure below illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 6 bytes, is located in the aligned doubleword containing the most-significant byte at 2. First, SDL stores the 6 most-significant bytes of the source register into these bytes in memory. Next, the complementary SDR instruction stores the remainder of *DW*.

Figure 4.25 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 ($vAddr_{2..0}$)—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 4.26 Bytes Stored by an SDL Instruction



Restrictions:

Availability:

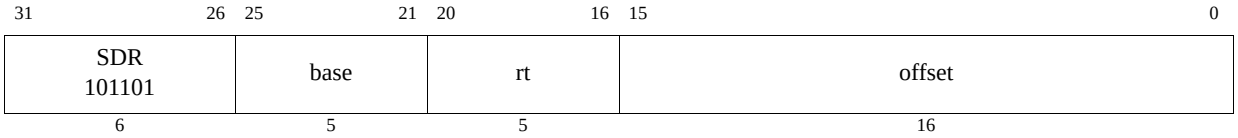
Release 6 removes the load/store-left/right family of instructions, and requires the system to support misaligned memory accesses.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
If BigEndianMem = 0 then
   pAddr ← pAddrPSIZE-1..3 || 03
endif
bytesel ← vAddr2..0 xor BigEndianCPU3
datadoubleword ← 056-8*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



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

Purpose: Store Doubleword Right

To store the least-significant part of a doubleword to an unaligned memory address

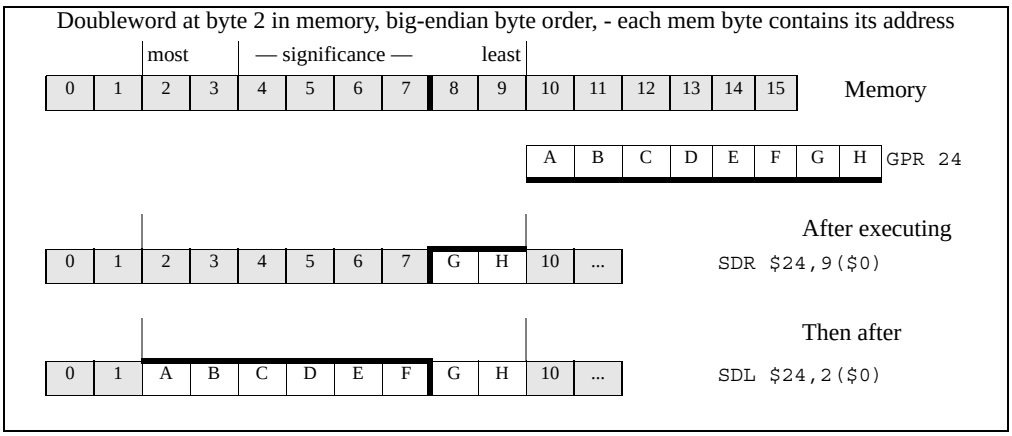
Description: `memory[GPR[base] + offset] ← Some_Bytes_From GPR[rt]`

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*.

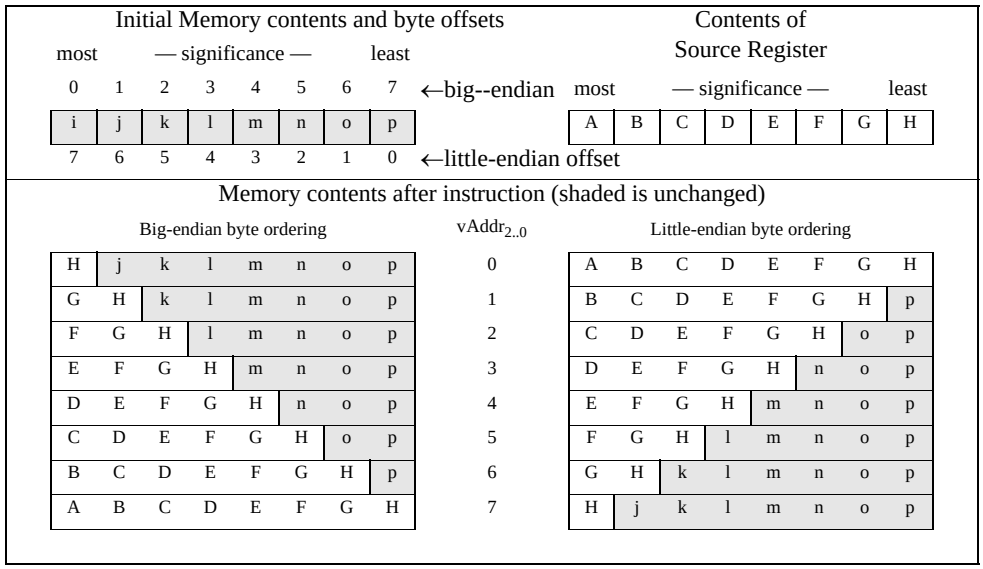
Figure 3-25 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 2 bytes, is located in the aligned doubleword containing the least-significant byte at 9. First, SDR stores the 2 least-significant bytes of the source register into these bytes in memory. Next, the complementary SDL stores the remainder of *DW*.

Figure 4.27 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 ($vAddr_{2..0}$)—and the current byte ordering mode of the processor (big- or little-endian). Figure 3-26 shows the bytes stored for every combination of offset and byte-ordering.

Figure 4.28 Bytes Stored by an SDR Instruction



Restrictions:

Availability:

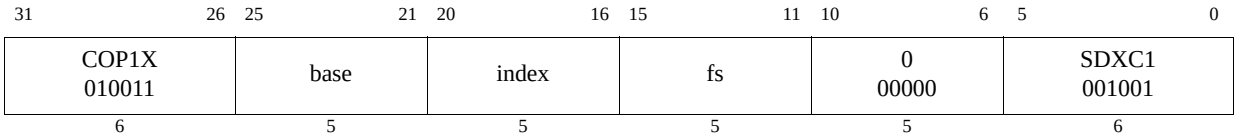
Release 6 removes the load/store-left/right family of instructions, and requires the system to support misaligned memory accesses.

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



Format: `SDXC1 fs, index(base)` **MIPS64**
MIPS32 Release 2

Purpose: Store Doubleword Indexed from Floating Point

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:

This instruction has been removed[1] in the MIPS32 Release 6 architecture.

Prior to MIPS32 Release 6, the following restrictions apply:

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

Availability:

Removed by Release 6.

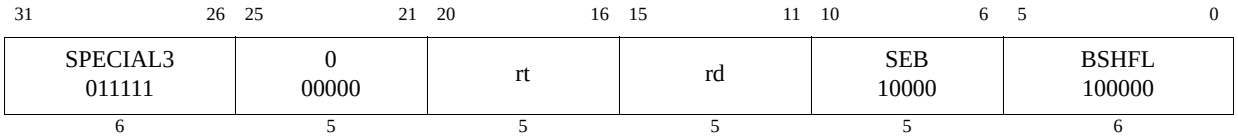
Operation:

```
vAddr ← GPR[base] + GPR[index]
if vAddr2..0 ≠ 03 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

1. "Removed by MIPS32 Release 6" means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Sign-Extend Byte

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 *r* 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[r]) then
   UNPREDICTABLE
endif
GPR[rd] ← sign_extend(GPR[rt]7..0)
```

Exceptions:

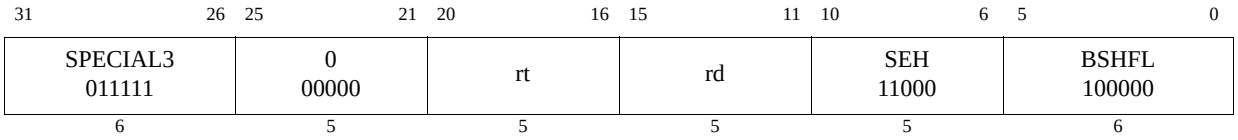
Reserved Instruction

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



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

Purpose: Sign-Extend Halfword

To sign-extend the least significant halfword of GPR *rt* and store the value into GPR *rd*.

Description: `GPR[rd] ← 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 *r* 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[r]) 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 */
```

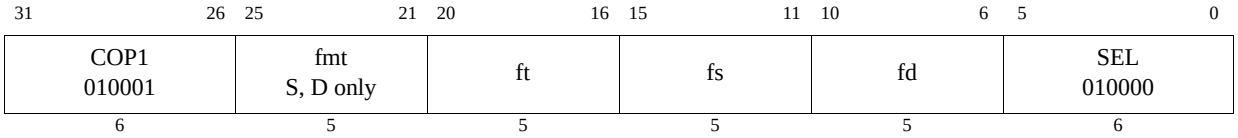
Zero-extended halfwords can be created by changing the SEH and SRA instructions to ANDI and SRL instructions, respectively.

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`

Expected Instruction	Function	Equivalent Instruction
`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



Format: `SEL.fmt`

`SEL.S fd,fs,ft` **MIPS32 Release 6**

`SEL.D fd,fs,ft` **MIPS32 Release 6**

Purpose: Select floating point values with FPR condition

Description: `fd ← fd.bit0 ? ft : fs`

SEL.fmt is a select operation, with a condition input in FPR `fd`, and 2 data inputs in FPRs `ft` and `fs`.

If the condition is true, the value of `ft` is written to `fd`.

If the condition is false, the value of `fs` is written to `fd`.

The condition input is specified by FPR `fd`, and is overwritten by the result.

The condition is true if and only if bit 0 of the condition input FPR `fd` is set. Other bits are ignored.

This instruction has floating point formats S and D, but these specify only the width of the operands. SEL.S can be used for 32-bit W data, and SEL.D can be used for 64 bit L data.

This instruction has no exception behavior. It does not trap on NaNs. It does not set the FPU Cause bits.

Restrictions:

Availability:

SEL.fmt is introduced by and required as of MIPS32 Release 6.

Special Considerations:

Only formats S and D are valid. Other format values may be used to encode other instructions. Unused format encodings are required to signal the Reserved Instruction exception.

Operation

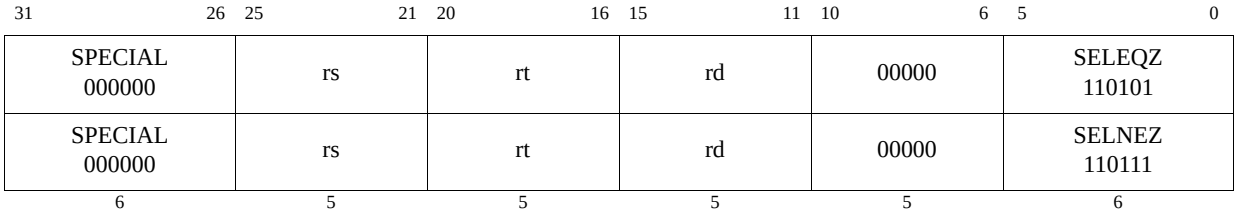
```
tmp ← ValueFPR(fd, UNINTERPRETED_WORD)
cond ← tmp.bit0
if cond then
   tmp ←ValueFPR(ft, fmt)
else
   tmp ←ValueFPR(fs, fmt)
endif
StoreFPR(fd, fmt, tmp)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	00000	SELEQZ 110101
SPECIAL 000000	rs	rt	rd	00000	SELNEZ 110111
6	5	5	5	5	6

Format: `SELEQZ SELNEZ`

`SELEQZ rd,rs,rt` **MIPS32 Release 6**

`SELNEZ rd,rs,rt` **MIPS32 Release 6**

Purpose: Select integer GPR value or zero

Description:

```
SELEQZ: rd ← areAnyBitsSet(rt) ? 0 : rs
SELNEZ: rd ← areAnyBitsSet(rt) ? rs: 0
```

SELEQZ is a select operation, with a condition input in GPR `rt`, one explicit data input in GPR `rs`, and implicit data input 0. The condition is true if and only if all bits in GPR `rt` are zero.

SELNEZ is a select operation, with a condition input in GPR `rt`, one explicit data input in GPR `rs`, and implicit data input 0. The condition is true if and only if any bit in GPR `rt` is nonzero

If the condition is true, the value of `rs` is written to `rd`.

If the condition is false, the zero written to `rd`.

This instruction operates on all GPRLEN bits of the CPU registers: i.e. all 32 bits on a 32-bit CPU, and all 64 bits on a 64-bit CPU. All GPRLEN bits of `rt` are tested.

Restrictions:

Availability:

SELEQZ is introduced by and required as of MIPS32 Release 6.

SELNEZ is introduced by and required as of MIPS32 Release 6.

Special Considerations:

Operation

```
SELNEZ: cond ← GPR[rt] ≠ 0
SELEQZ: cond ← GPR[rt] = 0
if cond then
   result ← GPR[rs]
else
   result ← GPR[rs]
endif
GPR[rd] ← result
```

Exceptions:

None

Programming Note:

MIPS32 Release 6 removes the pre-MIPS32 Release 6 instructions MOVZ and MOVN:

```
MOVZ:  if GPR[rt] = 0 then GPR[rd] ← GPR[rs]
MOVN:  if GPR[rt] ≠ 0 then GPR[rd] ← GPR[rs]
```

MOVZ can be emulated using MIPS32 Release 6 instructions as follows:

```
SELEQZ at, rs, rt
SELNEZ rd, rd, rt
OR rd, rd, at
```

Similarly MOVN:

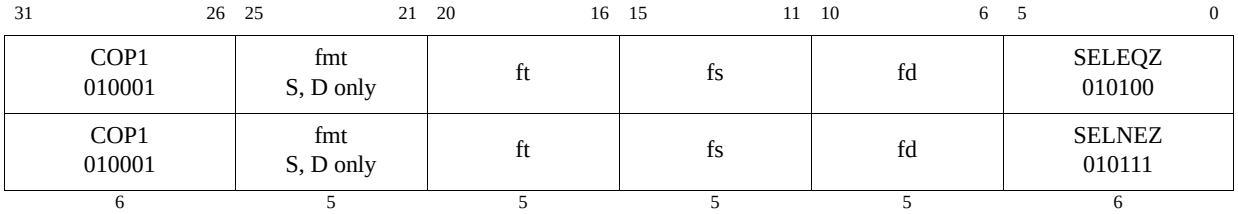
```
SELNEZ at, rs, rt
SELEQZ rd, rd, rt
OR rd, rd, at
```

The more general select operation requires 4 registers (1 output + 3 inputs (1 condition + 2 data)) can be expressed:

```
rD ← if rC then rA else rB
```

The more general select can be created using MIPS32 Release 6 instructions as follows:

```
    SELNEZ at, rB, rC
    SELNEZ rD, rA, rC
    OR rD, rD, at
```



31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt S, D only	ft	fs	fd	SELEQZ 010100
COP1 010001	fmt S, D only	ft	fs	fd	SELNEZ 010111
6	5	5	5	5	6

Format: `SELEQZ.fmt, SELNEQZ.fmt`

`SELEQZ.S fd,fs,ft` **MIPS32 Release 6**

`SELEQZ.D fd,fs,ft` **MIPS32 Release 6**

`SELNEZ.S fd,fs,ft` **MIPS32 Release 6**

`SELNEZ.D fd,fs,ft` **MIPS32 Release 6**

Purpose: Select floating point value or zero with FPR condition

Description:

```
SELEQZ.fmt: fd ← FPR[ft].bit0 ? 0 : fs
SELNEZ.fmt: fd ← FPR[ft].bit0 ? fs: 0
```

SELEQZ.fmt is a select operation, with a condition input in FPR `ft`, one explicit data input in FPR `fs`, and implicit data input 0. The condition is true if and only if bit 0 of FPR `ft` is zero.

SELNEZ.fmt is a select operation, with a condition input in FPR `ft`, one explicit data input in FPR `fs`, and implicit data input 0. The condition is true if and only if bit 0 in FPR `ft` is nonzero

If the condition is true, the value of `fs` is written to `fd`.

If the condition is false, the value that has all bits zero is written to `fd`.

This instruction has floating point formats S and D, but these specify only the width of the operands. Format S can be used for 32-bit W data, and format D can be used for 64 bit L data. The condition test is restricted to bit 0 of FPR `ft`. Other bits are ignored.

This instruction has no execution exception behavior. It does not trap on NaNs. It does not set the FPU Cause bits.

Restrictions:

FPR `fd` destination register bits beyond the format width are UNPREDICTABLE. E.g. if `fmt` is S, then `fd` bits 0-31 are defined, but bits 32 and above are UNPREDICTABLE. E.g. if `fmt` is D, then `fd` bits 0-63 are defined.

Availability:

SELEQZ.fmt is introduced by and required as of MIPS32 Release 6.

SELNEZ.fmt is introduced by and required as of MIPS32 Release 6.

Special Considerations:

Only formats S and D are valid. Other format values may be used to encode other instructions. Unused format encodings are required to signal the Reserved Instruction exception.

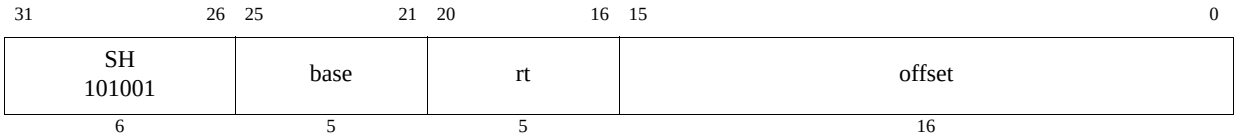
Operation

```
tmp ← ValueFPR(ft, UNINTERPRETED_WORD)
SELEQZ: cond ← tmp.bit0 = 0
SELNEZ: cond ← tmp.bit0 ≠ 0
if cond then
   tmp ← ValueFPR(fs, fmt)
else
   tmp ← 0 /* all bits set to zero */
endif
StoreFPR(fd, fmt, tmp)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:



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

Purpose: Store Halfword

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:

Prior to MIPS32 Release 6: The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

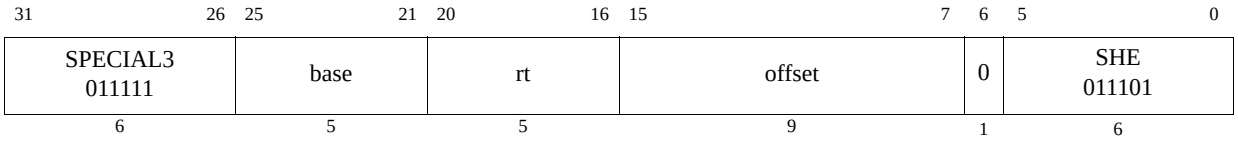
MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   if not MisalignedSupport()
      then SignalException(AddressError)
      else /* implementation dependent misalignment handling */
   endif
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian2 || 0))
bytesel ← vAddr2..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



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

Purpose: Store Halfword EVA

To store a halfword to user mode virtual address space when executing in kernel mode.

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

The SHE instruction functions in exactly the same fashion as the SH instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Restrictions:

Only usable in kernel mode when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Prior to MIPS32 Release 6: The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   if not MisalignedSupport()
      then SignalException(AddressError)
      else /* implementation dependent misalignment handling */
   endif
endif

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian2 || 0))
bytesel ← vAddr2..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

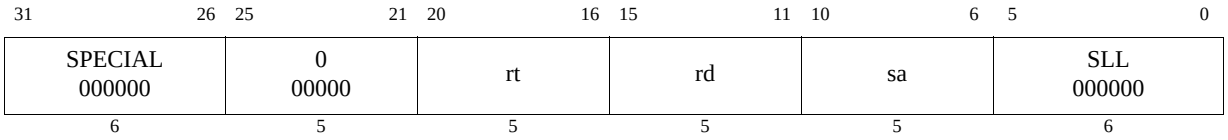
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



Format: `SLL rd, rt, sa` **MIPS32**

Purpose: Shift Word Left Logical

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:

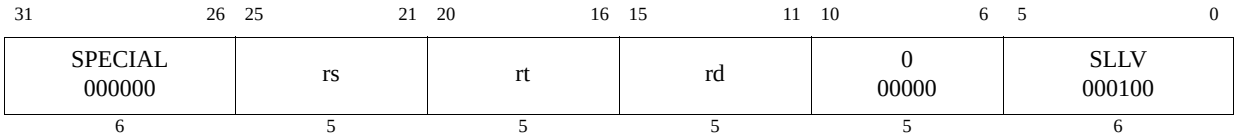
None

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: Shift Word Left Logical Variable

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:

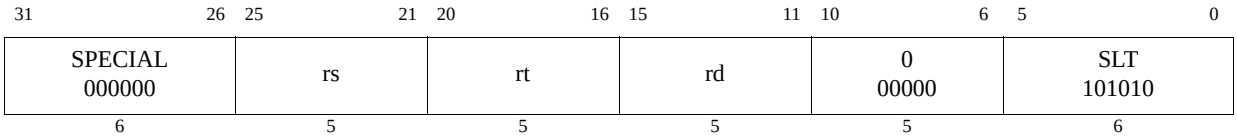
```
s ← GPR[rs]4..0
temp ← GPR[rt](31-s)..0 || 0s
GPR[rd] ← sign_extend(temp)
```

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.



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

Purpose: Set on Less Than

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:

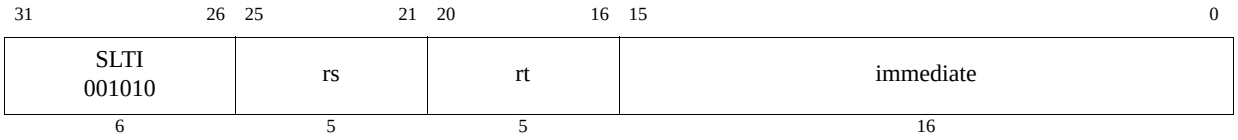
None

Operation:

```
if GPR[rs] < GPR[rt] then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None



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

Purpose: Set on Less Than Immediate

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:

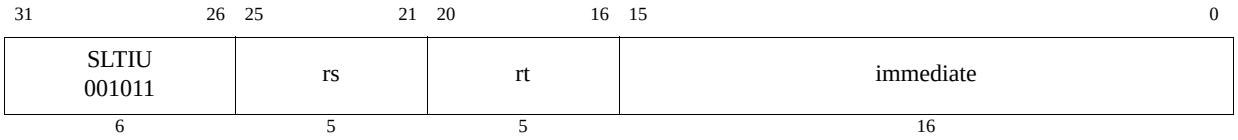
None

Operation:

```
if GPR[rs] < sign_extend(immediate) then
   GPR[rt] ← 0^(GPRLEN-1) || 1
else
   GPR[rt] ← 0^GPRLEN
endif
```

Exceptions:

None



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

Purpose: Set on Less Than Immediate Unsigned

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:

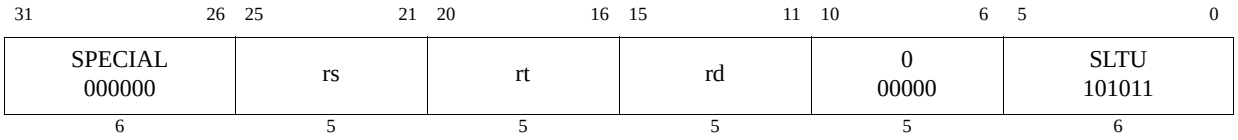
None

Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
   GPR[rt] ← 0^(GPRLEN-1) || 1
else
   GPR[rt] ← 0^GPRLEN
endif
```

Exceptions:

None



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

Purpose: Set on Less Than Unsigned

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:

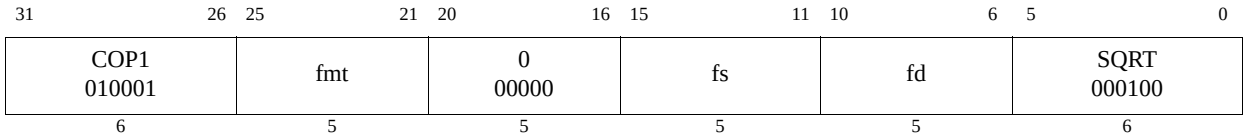
None

Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None



Format: `SQRT.fmt`

`SQRT.S fd, fs` **MIPS32**

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

Purpose: Floating Point Square Root

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:

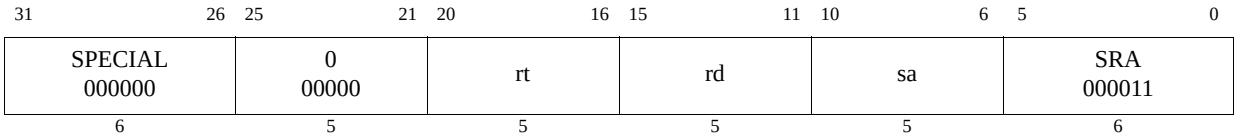
```
StoreFPR(fd, fmt, SquareRoot(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Inexact, Unimplemented Operation



Format: `SRA rd, rt, sa` **MIPS32**

Purpose: Shift Word Right Arithmetic

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:

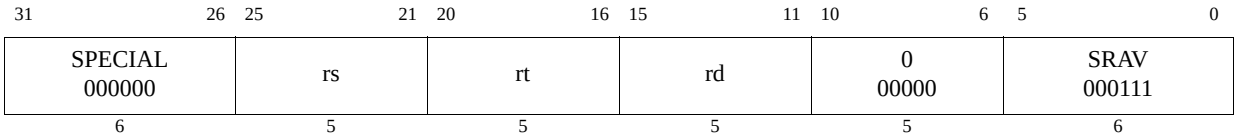
On 64-bit processors, if GPR *r* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[r]) then
   UNPREDICTALBE
endif
s ← sa
temp ← (GPR[rt]31)s || GPR[rt]31..s
GPR[rd] ← sign_extend(temp)
```

Exceptions:

None



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

Purpose: Shift Word Right Arithmetic Variable

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:

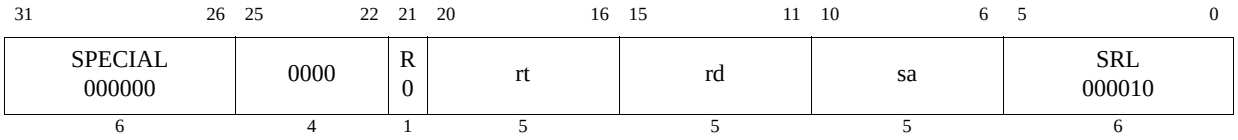
On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

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



Format: `SRL rd, rt, sa` **MIPS32**

Purpose: Shift Word Right Logical

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:

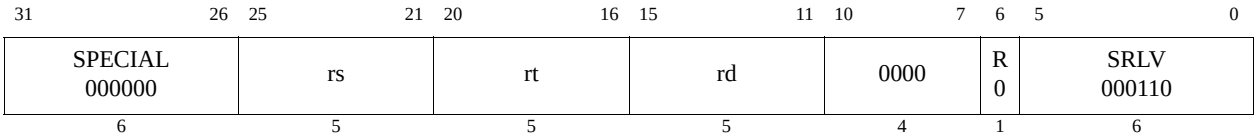
On 64-bit processors, if GPR *r* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[r]) then
   UNPREDICTABLE
endif
s ← sa
temp ← 0^s || GPR[rt]_31..s
GPR[rd] ← sign_extend(temp)
```

Exceptions:

None



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

Purpose: Shift Word Right Logical Variable

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:

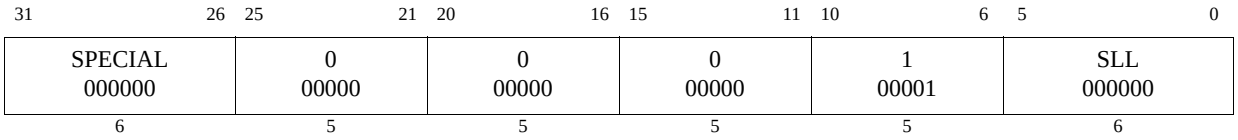
On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
s ← GPR[rs]4..0
temp ← 0s || GPR[rt]31..s
GPR[rd] ← sign_extend(temp)
```

Exceptions:

None



Format: `SSNOP` **MIPS32,**

Purpose: Superscalar No Operation

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

In MIPS32 Release 6 the special no-operation instruction SSNOP is deprecated: it behaves exactly the same as a conventional NOP. Its special behavior with respect to instruction issue is no longer guaranteed.

Assemblers targeting specifically Release 6 should reject the SSNOP instruction with an error.

Operation:

None

Exceptions:

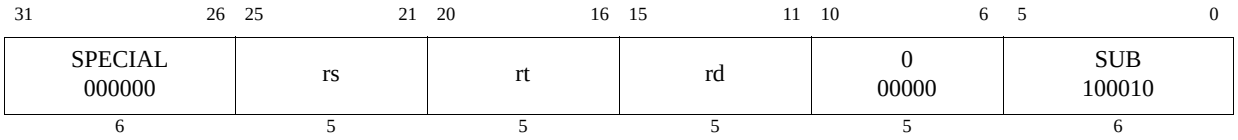
None

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.



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

Purpose: Subtract Word

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:

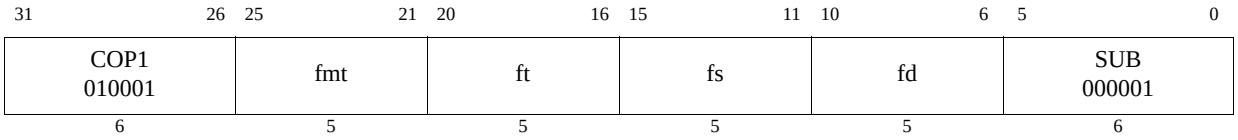
```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← (GPR[rs]31||GPR[rs]31..0) – (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← sign_extend(temp31..0)
endif
```

Exceptions:

Integer Overflow

Programming Notes:

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



Format: `SUB.fmt`

`SUB.S fd, fs, ft` **MIPS32**

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

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

Purpose: Floating Point Subtract

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 the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Availability:

This instruction has been removed in the Release 6 architecture.

Operation:

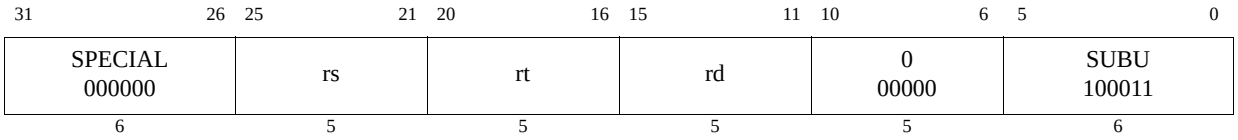
```
StoreFPR (fd, fmt, ValueFPR(fs, fmt) –fmt ValueFPR(ft, fmt))
```

CPU Exceptions:

Coprocessor Unusable, Reserved Instruction

FPU Exceptions:

Inexact, Overflow, Underflow, Invalid Op, Unimplemented Op



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

Purpose: Subtract Unsigned Word

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:

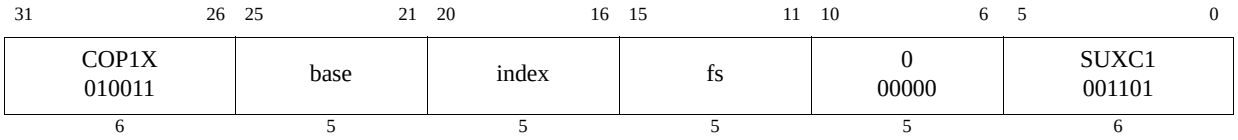
```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← GPR[rs] – GPR[rt]
GPR[rd] ← sign_extend(temp)
```

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.



Format: `SUXC1 fs, index(base)` **MIPS64, MIPS32 Release 2,**

Purpose: Store Doubleword Indexed Unaligned from Floating Point

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:

This instruction has been removed[1] in the MIPS32 Release 6 architecture.

Prior to MIPS32 Release 6, the following restrictions apply:

- The result of this instruction is **UNPREDICTABLE** if the processor is executing in the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.
-

Availability:

This instruction has been removed in the Release 6 architecture.

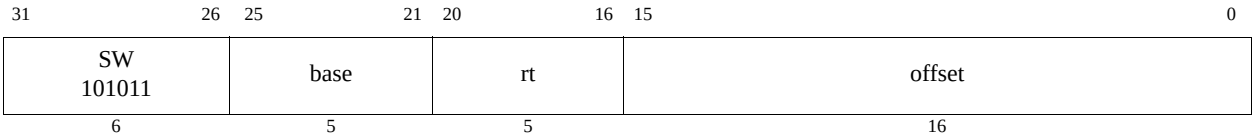
Operation:

```
vAddr ← (GPR[base]+GPR[index])63..3  || 03
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

1. "Removed by MIPS32 Release 6" means that implementations of MIPS32 Release 6 are required to signal the Reserved Instruction exception if there is no higher priority exception and if the instruction encoding has not been reused for a different instruction.



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

Purpose: Store Word

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:

Prior to MIPS32 Release 6: 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.

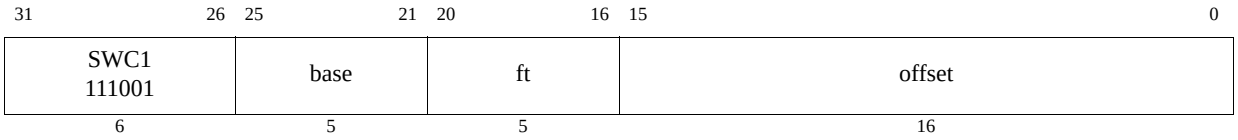
MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
   if not MisalignedSupport()
      then SignalException(AddressError)
      else /* implementation dependent misalignment handling */
   endif
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
datadoubleword ← GPR[rt]63-8*bytesel..0 || 08*bytesel
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch




```
SWC1 ft, offset(base)
```

MIPS32

Purpose: Store Word from Floating Point

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:

Prior to MIPS32 Release 6: An Address Error exception occurs if $EffectiveAddress_{1..0} \neq 0$ (not word-aligned).

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

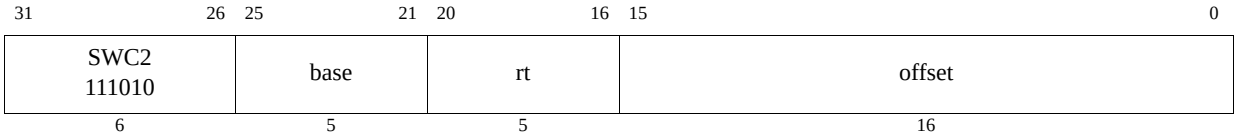
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
       if vAddr1..0 ≠ 0^3 then
           if not MisalignedSupport()
               then SignalException(AddressError)
               else /* implementation dependent misalignment handling */
           endif
       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)
```

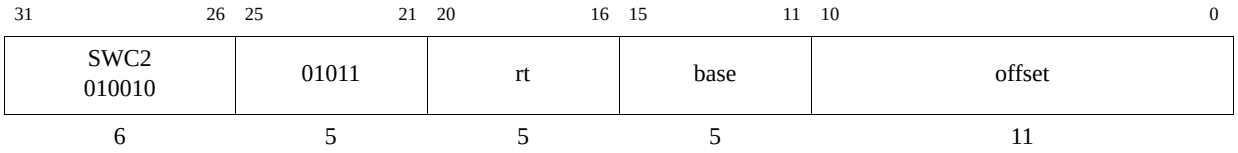
Exceptions:

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

MIPS32, (all release levels less than Release 6)



MIPS32 Release 6 (Release 6 and future)



Format: `SWC2 rt, offset(base)` **MIPS32**

Purpose: Store Word from Coprocessor 2

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:

Prior to MIPS32 Release 6: An Address Error exception occurs if $EffectiveAddress_{1..0} \neq 0$ (not word-aligned).

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

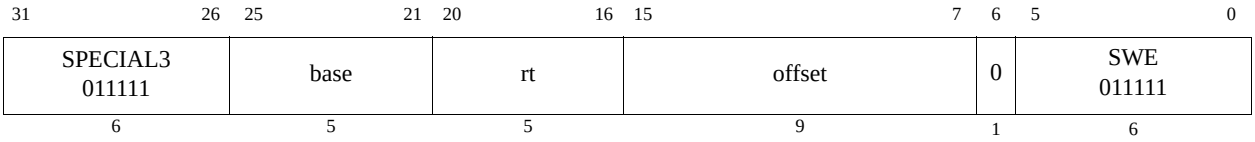
```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
          if not MisalignedSupport()
              then SignalException(AddressError)
              else /* implementation dependent misalignment handling */
          endif
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
datadoubleword ← CPR[2,rt,0]63-8*bytesel..0 || 08*bytesel
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Programming Notes:

As shown in the instruction drawing above, the MIPS Release 6 architecture implements an 11-bit offset, whereas all release levels lower than Release 6 of the MIPS architecture implement a 16-bit offset.



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

Purpose: Store Word EVA

To store a word to user mode virtual address space when executing in kernel mode.

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

The SWE instruction functions in exactly the same fashion as the SW instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Restrictions:

Only usable in kernel mode when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Prior to MIPS32 Release 6: 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.

MIPS32 Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate. See Appendix B, “Misaligned Memory Accesses” on page 768.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   if not MisalignedSupport()
      then SignalException(AddressError)
      else /* implementation dependent misalignment handling */
   endif
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

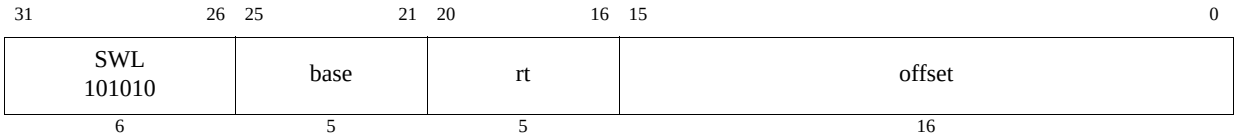
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



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

Purpose: Store Word Left

To store the most-significant part of a word to an unaligned memory address

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

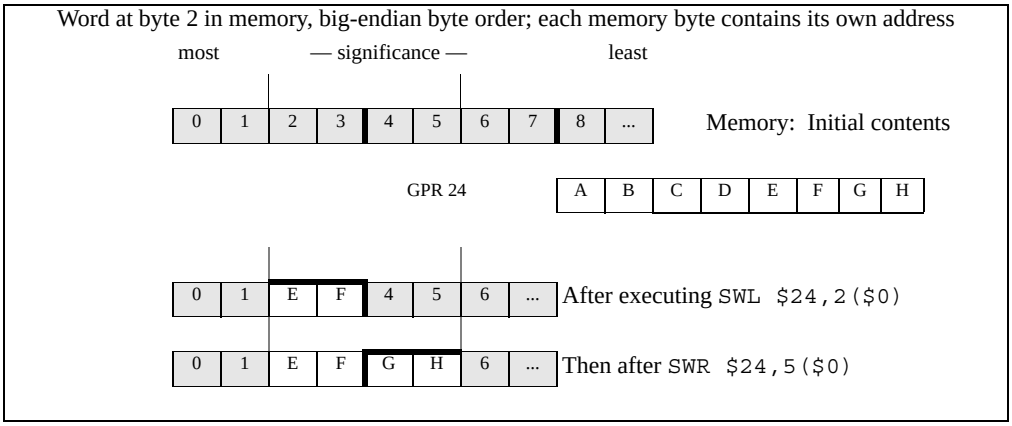
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

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.

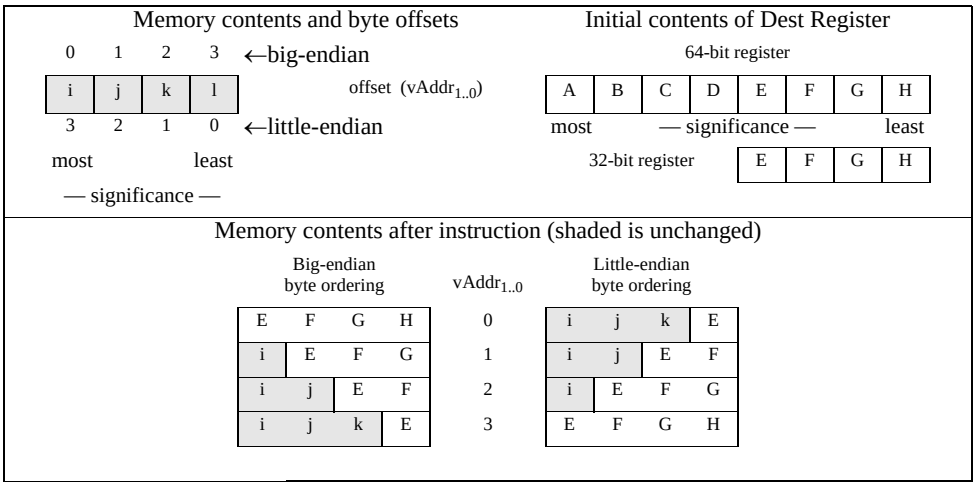
The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWL stores 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 4.29 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 ($vAddr_{1..0}$)—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 4.30 Bytes Stored by an SWL Instruction



Restrictions:

None.

Availability:

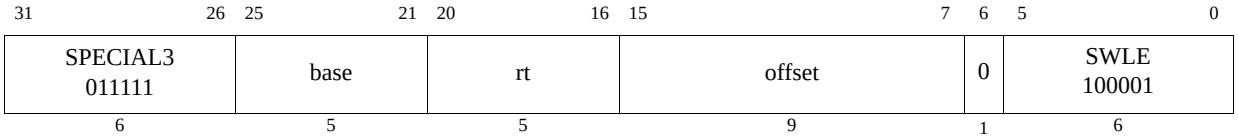
This instruction has been removed in the MIPS32 Release 6 architecture.

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



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

Purpose: Store Word Left EVA

To store the most-significant part of a word to an unaligned user mode virtual address while operating in kernel mode.

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

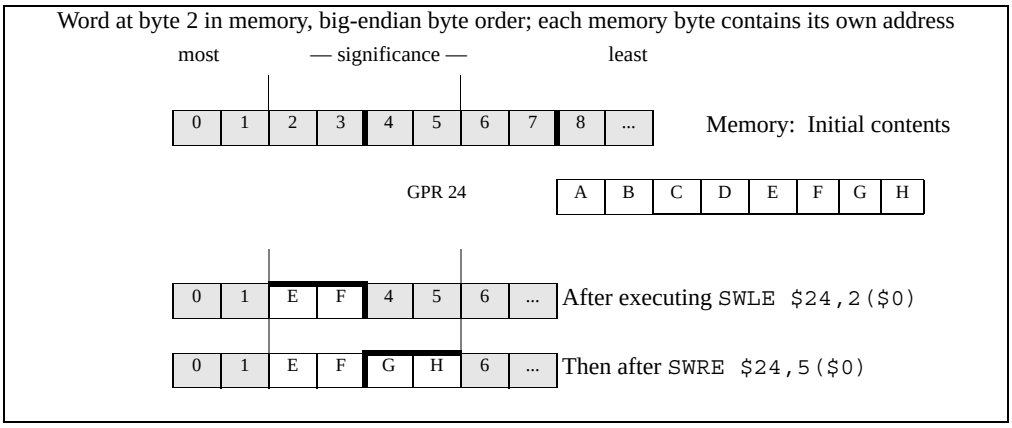
The 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

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, SWLE stores the most-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWRE stores the remainder of the unaligned word.

Figure 4.31 Unaligned Word Store Using SWLE and SWRE

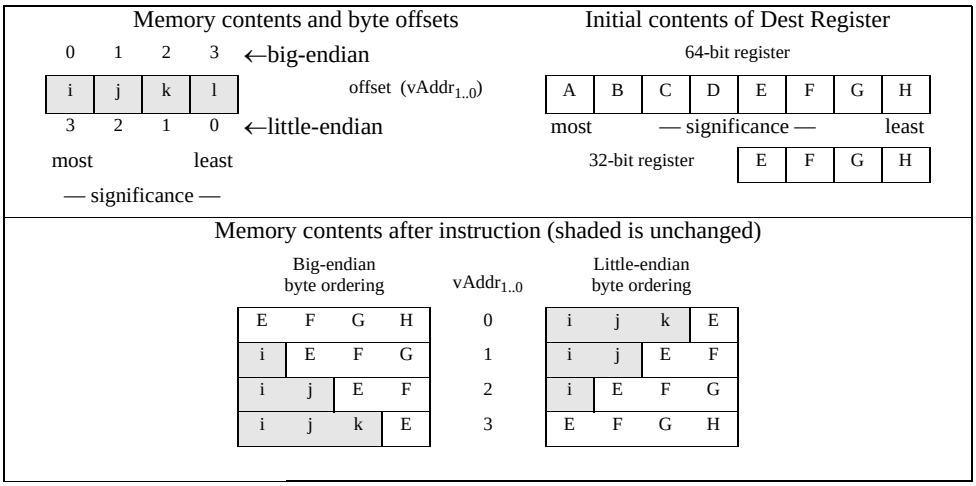


The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address ($vAddr_{1..0}$)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte ordering.

The SWLE instruction functions in exactly the same fashion as the SWL instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Figure 4.32 Bytes Stored by an SWLE Instruction



Restrictions:

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Availability:

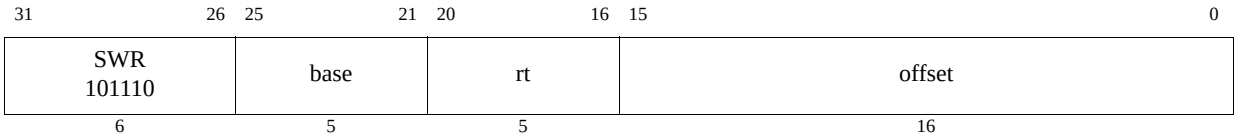
This instruction has been removed in the MIPS32 Release 6 architecture.

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, Reserved Instruction, Coprocessor Unusable



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

Purpose: Store Word Right

To store the least-significant part of a word to an unaligned memory address

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

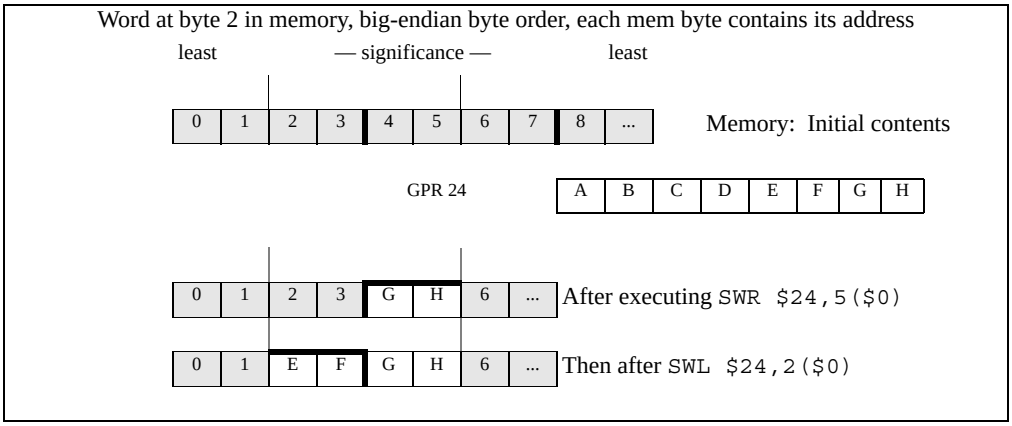
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

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.

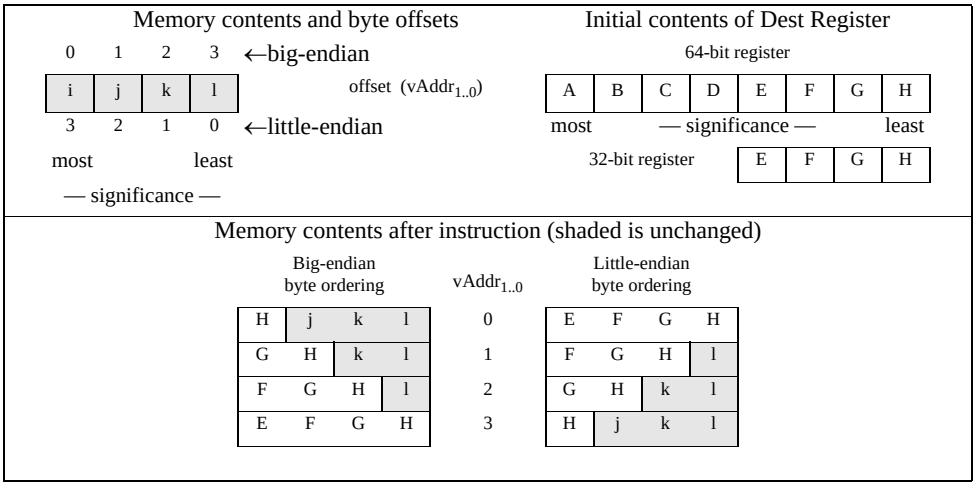
The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W,* 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR 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 4.33 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 ($vAddr_{1..0}$)—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 4.34 Bytes Stored by SWR Instruction



Restrictions:

None.

Availability:

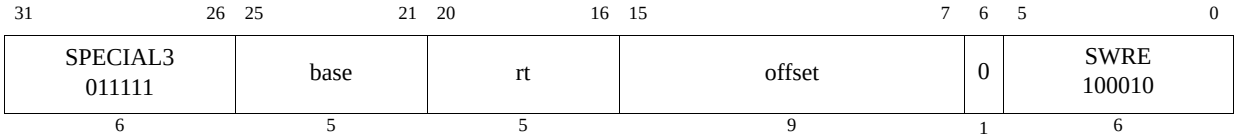
This instruction has been removed in the MIPS32 Release 6 architecture.

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



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

Purpose: Store Word Right EVA

To store the least-significant part of a word to an unaligned user mode virtual address while operating in kernel mode.

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

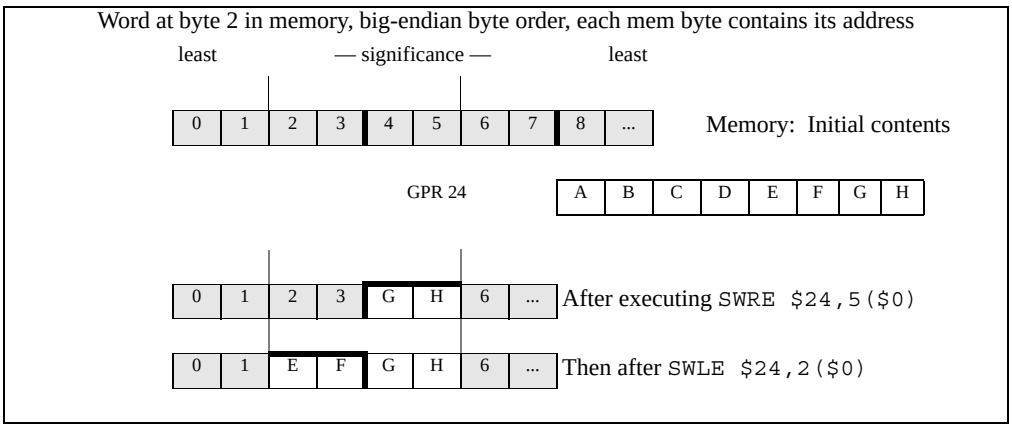
The 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

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, SWRE stores the least-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWLE stores the remainder of the unaligned word.

Figure 4.35 Unaligned Word Store Using SWRE and SWLE

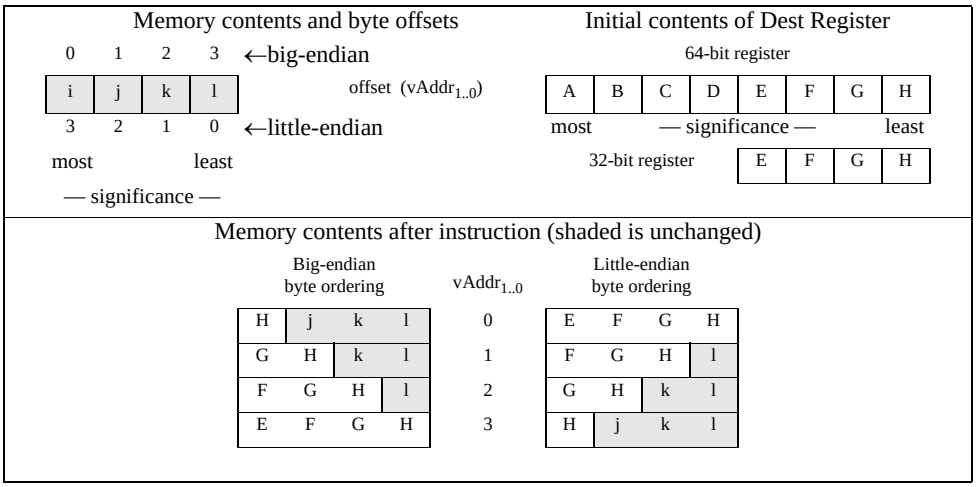


The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address ($vAddr_{1..0}$)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte-ordering.

The LWE instruction functions in exactly the same fashion as the LW instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Figure 4.36 Bytes Stored by SWRE Instruction



Restrictions:

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Availability:

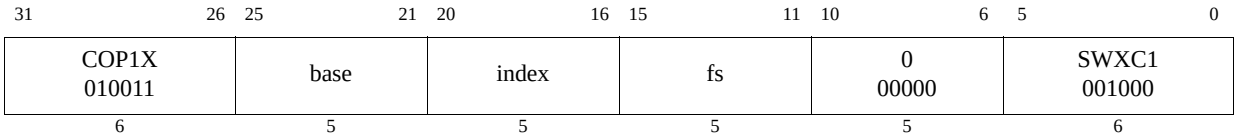
This instruction has been removed in the MIPS32 Release 6 architecture.

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, Coprocessor Unusable



Format: `SWXC1 fs, index(base)` **MIPS64MIPS32 Release 2**

Purpose: Store Word Indexed from Floating Point

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:

Prior to MIPS32 Release 6, the following restrictions apply:

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

Availability:

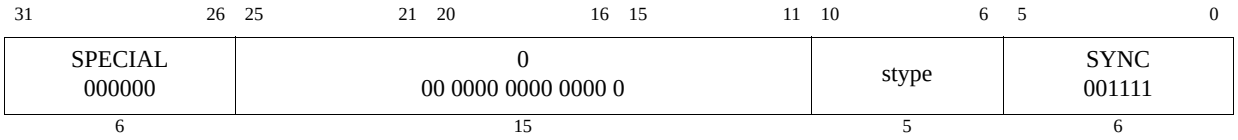
Removed by MIPS32 Release 6.

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(fs, 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**

`SYNC stype` **MIPS32**

Purpose: Synchronize Shared Memory

To order loads and stores for shared memory.

Description:

These types of ordering guarantees are available through the SYNC instruction:

- Completion Barriers
- Ordering Barriers

Simple Description for Completion Barrier:

- The barrier affects only *uncached* and *cached coherent* loads and stores.
- The specified memory instructions (loads or stores or both) that occur before the SYNC instruction must be completed before the specified memory instructions 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.

Detailed Description for Completion Barrier:

- Every synchronizable specified memory instruction (loads or stores or both) that occurs in the instruction stream before the SYNC instruction must be already globally performed before any synchronizable specified memory instructions that occur after the SYNC are allowed to be performed, with respect to any other processor or coherent I/O module.
- The barrier does not guarantee the order in which instruction fetches are performed.
- A stype value of zero will always be defined such that it performs the most complete set of synchronization operations that are defined.This means stype zero always does a completion barrier that affects both loads and stores preceding the SYNC instruction and both loads and stores that are subsequent to the SYNC instruction. Non-zero values of stype may be defined by the architecture or specific implementations to perform synchronization behaviors that are less complete than that of stype zero. If an implementation does not use one of these non-zero values to define a different synchronization behavior, then that non-zero value of stype must act the same as stype zero completion barrier. This allows software written for an implementation with a lighter-weight barrier to work on another implementation which only implements the stype zero completion barrier.
- A completion barrier 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 completion barrier is required on some implementations on entry to and exit from Debug Mode to guarantee that memory effects are handled correctly.

SYNC behavior when the stype field is zero:

- A completion barrier that affects preceding loads and stores and subsequent loads and stores.

Simple Description for Ordering Barrier:

- The barrier affects only *uncached* and *cached coherent* loads and stores.
- The specified memory instructions (loads or stores or both) that occur before the SYNC instruction must always be ordered before the specified memory instructions after the SYNC.
- Memory instructions which are ordered before other memory instructions are processed by the load/store datapath first before the other memory instructions.

Detailed Description for Ordering Barrier:

- Every synchronizable specified memory instruction (loads or stores or both) that occurs in the instruction stream before the SYNC instruction must reach a stage in the load/store datapath after which no instruction re-ordering is possible before any synchronizable specified memory instruction which occurs after the SYNC instruction in the instruction stream reaches the same stage in the load/store datapath.
- If any memory instruction before the SYNC instruction in program order, generates a memory request to the external memory and any memory instruction after the SYNC instruction in program order also generates a memory request to external memory, the memory request belonging to the older instruction must be globally performed before the time the memory request belonging to the younger instruction is globally performed.
- The barrier does not guarantee the order in which instruction fetches are performed.

As compared to the completion barrier, the ordering barrier is a lighter-weight operation as it does not require the specified instructions before the SYNC to be already completed. Instead it only requires that those specified instructions which are subsequent to the SYNC in the instruction stream are never re-ordered for processing ahead of the specified instructions which are before the SYNC in the instruction stream. This potentially reduces how many cycles the barrier instruction must stall before it completes.

The Acquire and Release barrier types are used to minimize the memory orderings that must be maintained and still have software synchronization work.

Implementations that do not use any of the non-zero values of stype to define different barriers, such as ordering barriers, must make those stype values act the same as stype zero.

For the purposes of this description, the CACHE, PREF and PREFX instructions are treated as loads and stores. That is, these instructions and the memory transactions sourced by these instructions obey the ordering and completion rules of the SYNC instruction.

Table 4.16 lists the available completion barrier and ordering barriers behaviors that can be specified using the stype field..

Table 4.16 Encodings of the Bits[10:6] of the SYNC instruction; the SType Field

Code	Name	Older instructions which must reach the load/store ordering point before the SYNC instruction completes.	Younger instructions which must reach the load/store ordering point only after the SYNC instruction completes.	Older instructions which must be globally performed when the SYNC instruction completes	Compliance
0x0	SYNC or SYNC 0	Loads, Stores	Loads, Stores	Loads, Stores	Required
0x4	SYNC_WMB or SYNC 4	Stores	Stores		Optional
0x10	SYNC_MB or SYNC 16	Loads, Stores	Loads, Stores		Optional
0x11	SYNC_ACQUIRE or SYNC 17	Loads	Loads, Stores		Optional
0x12	SYNC_RELEASE or SYNC 18	Loads, Stores	Stores		Optional
0x13	SYNC_RMB or SYNC 19	Loads	Loads		Optional
0x1-0x3, 0x5-0xF					Implementation-Specific and Vendor Specific Sync Types
0x14 - 0x1F	RESERVED				Reserved for MIPS Technologies for future extension of the architecture.

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*.

Load/Store Datapath: The portion of the processor which handles the load/store data requests coming from the processor pipeline and processes those requests within the cache and memory system hierarchy.

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.)

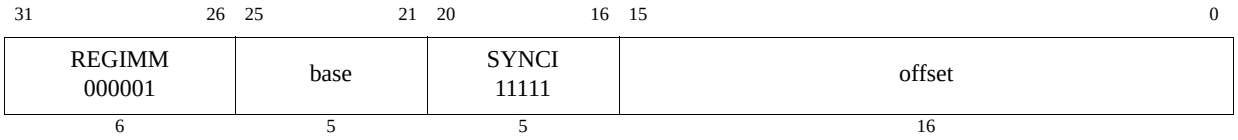
The behavior of a load or store using one memory access type is **UNPREDICTABLE** if a load or store was previously made to the same physical location using a different memory access type. The presence of a SYNC between the references does not alter this behavior.

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.

```
# 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
```

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.

Software written to use a SYNC instruction with a non-zero stype value, expecting one type of barrier behavior, should only be run on hardware that actually implements the expected barrier behavior for that non-zero stype value or on hardware which implements a superset of the behavior expected by the software for that stype value. If the hardware does not perform the barrier behavior expected by the software, the system may fail.



Format: `SYNCI offset(base)` **MIPS32 Release 2**

Purpose: Synchronize Caches to Make Instruction Writes Effective

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. This instruction never causes Execute-Inhibit nor Read-Inhibit exceptions.

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.In multiprocessor implementations where instruction caches are not coherently maintained by hardware, the SYNCI instruction may optionally affect all coherent icaches within the system. If the effective address uses a coherent Cacheability and Coherency Attribute (CCA), then the operation may be *globalized*, meaning it is broadcast to all of the coherent instruction caches within the system. If the effective address does not use one of the coherent CCAs, there is no broadcast of the SYNCI operation. If multiple levels of caches are to be affected by one SYNCI instruction, all of the affected cache levels must be processed in the same manner - either all affected cache levels use the globalized behavior or all affected cache levels use the non-globalized behavior.

In multiprocessor implementations where instruction caches are coherently maintained by hardware, the SYNCI instruction should behave as a NOP instruction.

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 previously 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 on the current processor at a minimum. It is implementation specific whether it affects

the caches on other processors in a multiprocessor 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 */
```

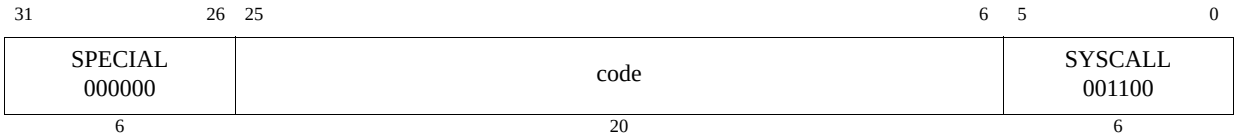
Exceptions:

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
 */

   beq    a1, zero, 20f       /* If size==0, */
   nop                        /*   branch around */
   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 */
   addu   a0, a0, v0          /* Add step size in delay slot */
                              /* (daddu for 64-bit addressing) */
   sltu   v1, a0, a1          /* Compare current with end address */
   bne    v1, zero, 10b       /* Branch if more to do */
   nop                        /*   branch around */
   sync                       /* Clear memory hazards */
20:jr.hb  ra                  /* Return, clearing instruction hazards */
   nop
```



Format: `SYSCALL` **MIPS32**

Purpose: System Call

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:

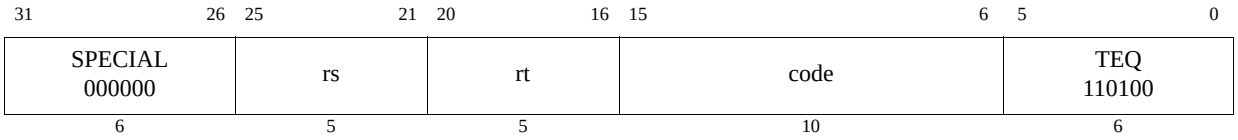
None

Operation:

```
SignalException(SystemCall)
```

Exceptions:

System Call



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

Purpose: Trap if Equal

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:

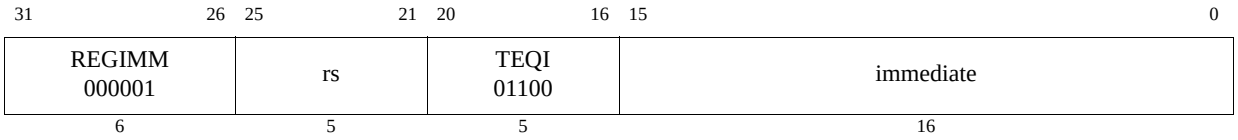
None

Operation:

```
if GPR[rs] = GPR[rt] then
   SignalException(Trap)
endif
```

Exceptions:

Trap



Format: `TEQI rs, immediate` **MIPS32,**

Purpose: Trap if Equal Immediate

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

Availability:

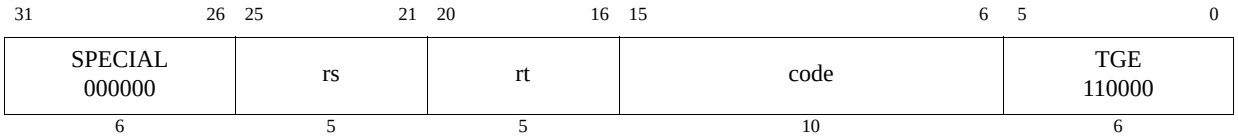
This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

```
if GPR[rs] = sign_extend(immediate) then
       SignalException(Trap)
endif
```

Exceptions:

Trap



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

Purpose: Trap if Greater or Equal

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:

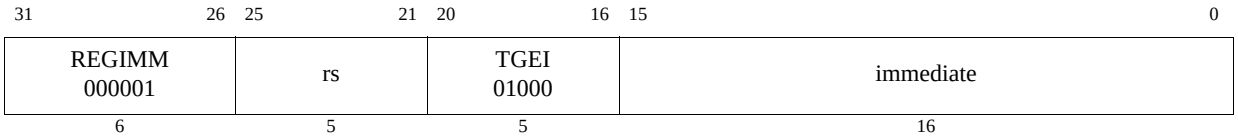
None

Operation:

```
if GPR[rs] ≥ GPR[rt] then
   SignalException(Trap)
endif
```

Exceptions:

Trap



Format: `TGEI rs, immediate` **MIPS32,**

Purpose: Trap if Greater or Equal Immediate

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

Availability:

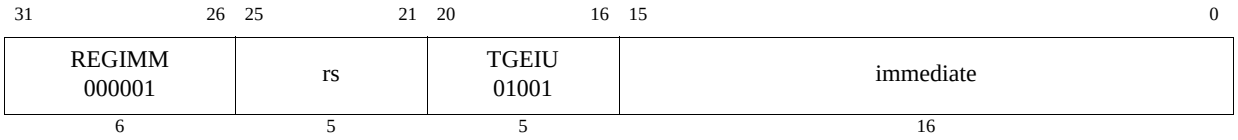
This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

```
if GPR[rs] ≥ sign_extend(immediate) then
   SignalException(Trap)
endif
```

Exceptions:

Trap



Format: `TGEIU rs, immediate` **MIPS32,**

Purpose: Trap if Greater or Equal Immediate Unsigned

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

Availability:

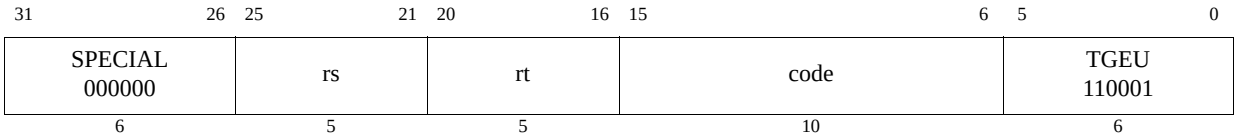
This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

```
if (0 || GPR[rs]) ≥ (0 || sign_extend(immediate)) then
   SignalException(Trap)
endif
```

Exceptions:

Trap



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

Purpose: Trap if Greater or Equal Unsigned

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:

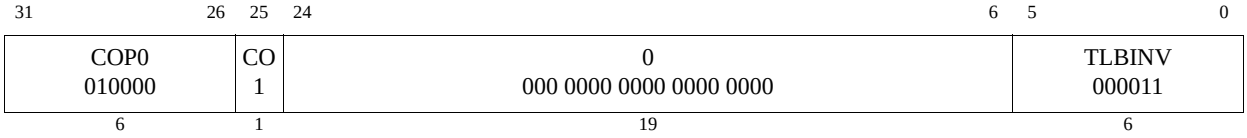
None

Operation:

```
if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
   SignalException(Trap)
endif
```

Exceptions:

Trap



Format: `TLBINV` **MIPS32**

Purpose: TLB Invalidate

TLBINV invalidates a set of TLB entries based on ASID and Index match. The virtual address is ignored in the entry match. TLB entries which have their G bit set to 1 are not modified.

Description:

On execution of the TLBINV instruction, the set of TLB entries with matching ASID are marked invalid, excluding those TLB entries which have their G bit set to 1.

The *EntryHI*$_{ASID}$ field has to be set to the appropriate ASID value before executing the TLBINV instruction.

Behavior of the TLBINV instruction applies to all applicable TLB entries and is unaffected by the setting of the *Wired* register.

For JTLB-based MMU (*Config*$_{MT}$=1):

All matching entries in the JTLB are invalidated. *Index* is unused.

For VTLB/FTLB -based MMU (*Config*$_{MT}$=4):

A TLBINV with *Index* set in VTLB range causes all matching entries in the VTLB to be invalidated.

A TLBINV with *Index* set in FTLB range causes all matching entries in the single corresponding FTLB set to be invalidated.

If TLB invalidate walk is implemented in software (*Config4*$_{IE}$=2), then software must do these steps to flush the entire MMU:

1. one TLBINV instruction is executed with an index in VTLB range (invalidates all matching VTLB entries)
2. a TLBINV instruction is executed for each FTLB set (invalidates all matching entries in FTLB set)

If TLB invalidate walk is implemented in hardware (*Config4*$_{IE}$=3), then software must do these steps to flush the entire MMU:

1. one TLBINV instruction is executed (invalidates all matching entries in both FTLB & VTLB). In this case, *Index* is unused.

Restrictions:

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of available TLB entries (For the case of *Config*$_{MT}$=4).

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

Availability:

Implementation of the TLBINV instruction is optional. The implementation of this instruction is indicated by the IE field in *Config4*.

Implementation of $EntryHI_{EHINV}$ field is required for implementation of TLBINV instruction.

Prior to MIPS32 Release 6, support for TLBINV is recommended for implementations supporting VTLB/FTLB type of MMU. Release 6 (and subsequent releases) support for TLBINV is required for implementations supporting VTLB/FTLB type of MMU.

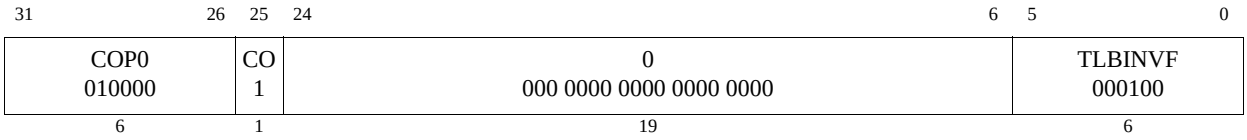
For MIPS32 Release 6 processors that include a Block Address Translation (BAT) or Fixed Mapping (FM) MMU ($Config_{MT}$ = 2 or 3), the operation of this instruction causes a Reserved Instruction exception (RI).

Operation:

```
if ( Config_MT=1 or (Config_MT=4 & Config4_IE=2 & Index ≤ Config1_MMU_SIZE-1))
    startnum ← 0
    endnum ← Config1_MMU_SIZE-1
endif
// treating VTLB and FTLB as one array
if (Config_MT=4 & Config4_IE=2 & Index > Config1_MMU_SIZE-1)
    startnum ← start of selected FTLB set // implementation specific
    endnum ← end of selected FTLB set - 1 //implementation specifc
endif

if (Config_MT=4 & Config4_IE=3))
    startnum ← 0
    endnum ← Config1_MMU_SIZE-1 + ((Config4_FTLBWays + 2) * Config4_FTLBSets)
endif

for (i = startnum to endnum)
    if (TLB[i].ASID = EntryHi_ASID & TLB[i].G = 0)
        TLB[i]_VPN2_invalid ← 1
    endif
endfor
```

Exceptions:

Coprocessor Unusable



Format: `TLBINVF` **MIPS32**

Purpose: TLB Invalidate Flush

TLBINVF invalidates a set of TLB entries based on *Index* match. The virtual address and ASID are ignored in the entry match.

Description:

On execution of the TLBINVF instruction, all entries within range of *Index* are invalidated.

Behavior of the TLBINVF instruction applies to all applicable TLB entries and is unaffected by the setting of the *Wired* register.

For JTLB-based MMU (*Config*$_{MT}$=1):

TLBINVF causes all entries in the JTLB to be invalidated. *Index* is unused.

For VTLB/FTLB-based MMU (*Config*$_{MT}$=4):

TLBINVF with *Index* in VTLB range causes all entries in the VTLB to be invalidated.

TLBINVF with *Index* in FTLB range causes all entries in the single corresponding set in the FTLB to be invalidated.

If TLB invalidate walk is implemented in software (*Config4*$_{IE}$=2), then software must do these steps to flush the entire MMU:

1. one TLBINV instruction is executed with an index in VTLB range (invalidates all VTLB entries)
2. a TLBINV instruction is executed for each FTLB set (invalidates all entries in FTLB set)

If TLB invalidate walk is implemented in hardware (*Config4*$_{IE}$=3), then software must do these steps to flush the entire MMU:

1. one TLBINV instruction is executed (invalidates all entries in both FTLB & VTLB). In this case, *Index* is unused.

Restrictions:

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of available TLB entries (*Config4*$_{IE}$=2).

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

Availability:

Implementation of the TLBINV instruction is optional. The implementation of this instruction is indicated by the IE field in *Config4*.

Implementation of *EntryHI*$_{EHINV}$ field is required for implementation of TLBINV instruction.

Prior to MIPS32 Release 6, support for TLBINV is recommended for implementations supporting VTLB/FTLB type of MMU. Release 6 (and subsequent releases) support for TLBINV is required for implementations supporting VTLB/FTLB type of MMU.

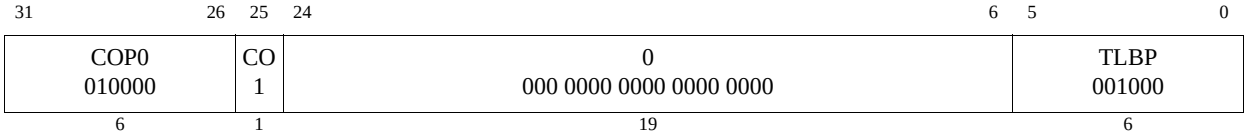
For MIPS32 Release 6 processors that include a Block Address Translation (BAT) or Fixed Mapping (FM) MMU (Config$_{MT}$ = 2 or 3), the operation of this instruction causes a Reserved Instruction exception (RI).

Operation:

```
if ( Config_MT=1 or (Config_MT=4 & Config4_IE=2 & Index ≤ Config1_MMU_SIZE-1))
   startnum ← 0
   endnum ← Config1_MMU_SIZE-1
endif
// treating VTLB and FTLB as one array
if (Config_MT=4 & Config4_IE=2 & Index > Config1_MMU_SIZE-1)
   startnum ← start of selected FTLB set // implementation specific
   endnum ← end of selected FTLB set - 1 //implementation specifc
endif

if (Config_MT=4 & Config4_IE=3))
   startnum ← 0
   endnum ← Config1_MMU_SIZE-1 +((Config4_FTLBWays + 2) * Config4_FTLBSets)
endif

for (i = startnum to endnum)
   TLB[i]_VPN2_invalid ← 1
endfor
```

Exceptions:

Coprocessor Unusable



Format: `TLBP` **MIPS32**

Purpose: Probe TLB for Matching Entry

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. In Release 3 of the Architecture, multiple TLB matches may be reported on either TLB write or TLB probe.

Restrictions:

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

For MIPS32 Release 6 processors that include a Block Address Translation (BAT) or Fixed Mapping (FM) MMU ($Config_{MT}$ = 2 or 3), the operation of this instruction causes a Reserved Instruction exception (RI).

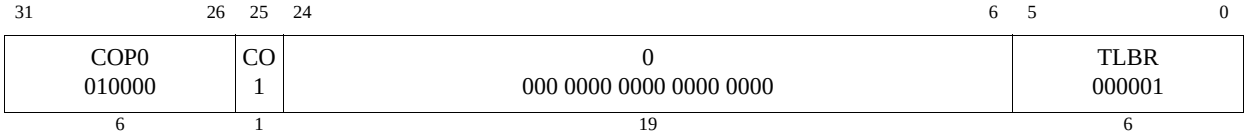
Operation:

```
Index ← 1 || UNPREDICTABLE31
for i in 0...TLBEntries-1
    if ((TLB[i]VPN2 and not (TLB[i]Mask)) =
                (EntryHiVPN2 and not (TLB[i]Mask))) and
        (TLB[i]R = EntryHiR) and
        ((TLB[i]G = 1) or (TLB[i]ASID = EntryHiASID)) then
        Index ← i
    endif
endfor
```

Exceptions:

Coprocessor Unusable

Machine Check



Format: `TLBR` **MIPS32**

Purpose: Read Indexed TLB Entry

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. In Release 3 of the Architecture, multiple TLB matches may be detected on a TLBR.

In an implementation supporting TLB entry invalidation ($Config4_{IE}$ = 2 or $Config4_{IE}$ = 3), reading an invalidated TLB entry causes 0 to be written to *EntryHi*, *EntryLo0*, *EntryLo1* registers and the $PageMask_{MASK}$ register field.

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 have those 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.

For MIPS32 Release 6 processors that include a Block Address Translation (BAT) or Fixed Mapping (FM) MMU ($Config_{MT}$ = 2 or 3), the operation of this instruction causes a Reserved Instruction exception (RI).

Operation:

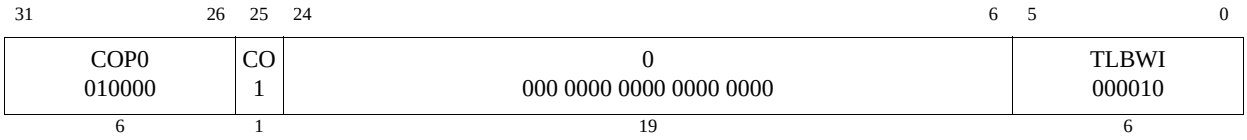
```
i ← Index
if i > (TLBEntries - 1) then
    UNDEFINED
endif
if ( (Config4IE = 2 or Config4IE = 3) and TLB[i]VPN2_invalid = 1) then
    PagemaskMask ← 0
```

```
    EntryHi ← 0
    EntryLo1 ← 0
    EntryLo0 ← 0
    EntryHi_EHINV ← 1
else
    PageMask_Mask ← TLB[i]_Mask
    EntryHi ← TLB[i]_R || 0^Fill ||
              (TLB[i]_VPN2 and not TLB[i]_Mask) || # Masking implem dependent
              0^5 || TLB[i]_ASID
    EntryLo1 ← 0^Fill ||
              (TLB[i]_PFN1 and not TLB[i]_Mask) || # Masking mplem 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 mplem dependent
           TLB[i]_C0 || TLB[i]_D0 || TLB[i]_V0 || TLB[i]_G
endif
```

Exceptions:

Coprocessor Unusable

Machine Check



Format: `TLBWI` **MIPS32**

Purpose: Write Indexed TLB Entry

To write or invalidate a TLB entry indexed by the *Index* register.

Description:

If $Config4_{IE} < 2$ or $EntryHi_{EHINV}=0$:

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.

If $Config4_{IE} > 1$ and $EntryHi_{EHINV}=1$:

The TLB entry pointed to by the Index register has its VPN2 field marked as invalid. This causes the entry to be ignored on TLB matches for memory accesses. No Machine Check is generated.

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.

For MIPS32 Release 6 processors that include a Block Address Translation (BAT) or Fixed Mapping (FM) MMU ($Config_{MT}$ = 2 or 3), the operation of this instruction causes a Reserved Instruction exception (RI).

Operation:

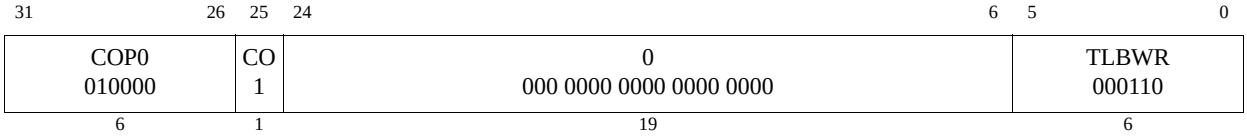
```
i ← Index
if (Config4IE = 2 or Config4IE = 3) then
   TLB[i]VPN2_invalid ← 0
   if ( EntryHIEHINV=1 ) then
      TLB[i]VPN2_invalid ← 1
```

```
        break
    endif
endif
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



Format: `TLBWR` **MIPS32**

Purpose: Write Random TLB Entry

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.

For MIPS32 Release 6 processors that include a Block Address Translation (BAT) or Fixed Mapping (FM) MMU ($Config_{MT}$ = 2 or 3), the operation of this instruction causes a Reserved Instruction exception (RI).

Operation:

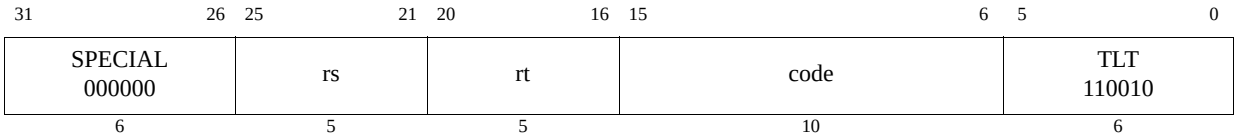
```
i ← Random
if (Config4IE = 2 or Config4IE = 3) then
   TLB[i]VPN2_invalid ← 0
   endif
TLB[i]Mask ← PageMaskMask
TLB[i]R ← EntryHiR
TLB[i]VPN2 ← EntryHiVPN2 and not PageMaskMask # Implementation dependent
TLB[i]ASID ← EntryHiASID
TLB[i]G ← EntryLo1G and EntryLo0G
TLB[i]PFN1 ← EntryLo1PFN and not 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



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

Purpose: Trap if Less Than

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:

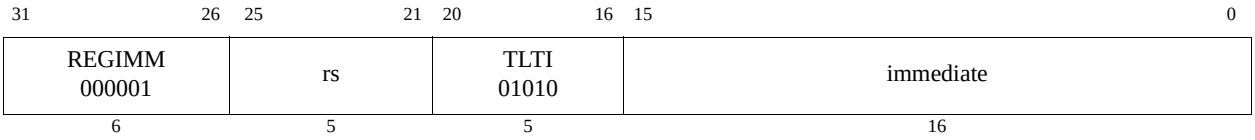
None

Operation:

```
if GPR[rs] < GPR[rt] then
   SignalException(Trap)
endif
```

Exceptions:

Trap



Format: `TLTI rs, immediate` **MIPS32,**

Purpose: Trap if Less Than Immediate

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

Availability:

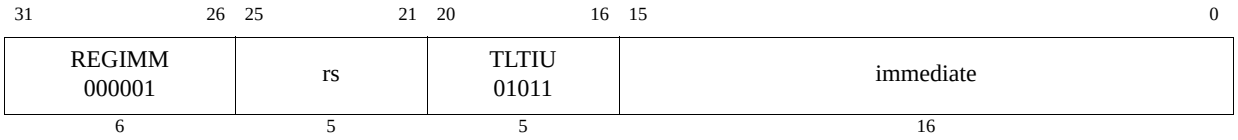
This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

```
if GPR[rs] < sign_extend(immediate) then
   SignalException(Trap)
endif
```

Exceptions:

Trap



Format: `TLTIU rs, immediate` **MIPS32,**

Purpose: Trap if Less Than Immediate Unsigned

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

Availability:

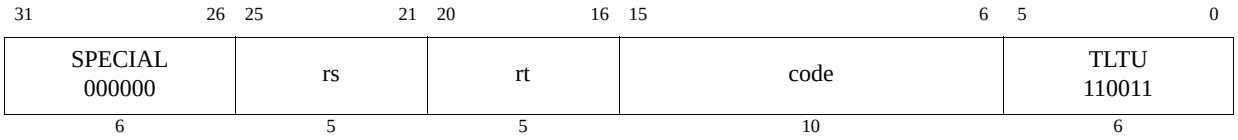
This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
   SignalException(Trap)
endif
```

Exceptions:

Trap



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

Purpose: Trap if Less Than Unsigned

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:

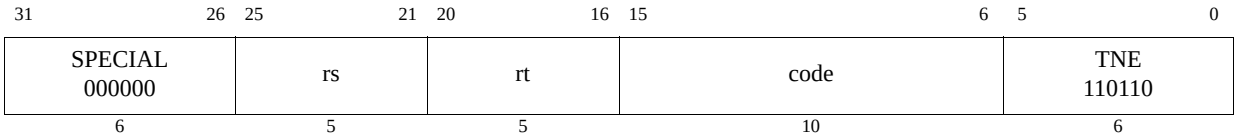
None

Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
   SignalException(Trap)
endif
```

Exceptions:

Trap



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

Purpose: Trap if Not Equal

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:

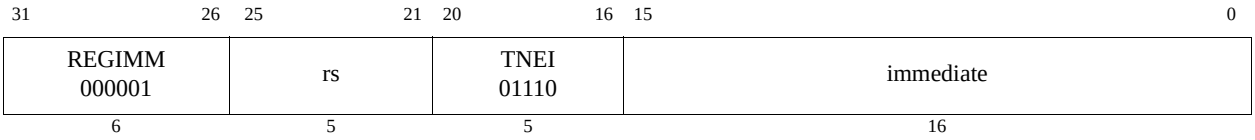
None

Operation:

```
if GPR[rs] ≠ GPR[rt] then
   SignalException(Trap)
endif
```

Exceptions:

Trap



Format: `TNEI rs, immediate` **MIPS32,**

Purpose: Trap if Not Equal Immediate

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

Availability:

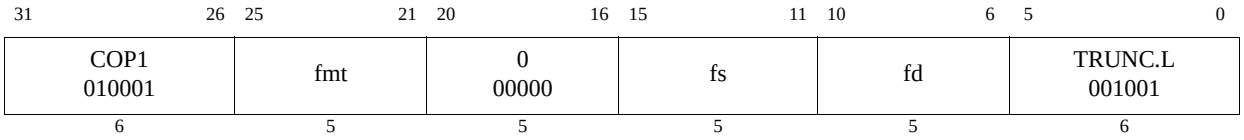
This instruction has been removed in the MIPS32 Release 6 architecture.

Operation:

```
if GPR[rs] ≠ sign_extend(immediate) then
   SignalException(Trap)
endif
```

Exceptions:

Trap



Format: `TRUNC.L.fmt`

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

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

Purpose: Floating Point Truncate to Long Fixed Point

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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

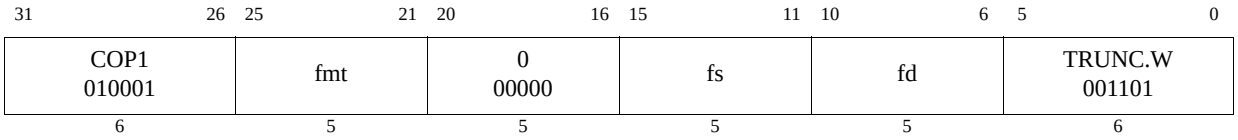
```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact



Format: `TRUNC.W.fmt`

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

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

Purpose: Floating Point Truncate to Word Fixed Point

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:

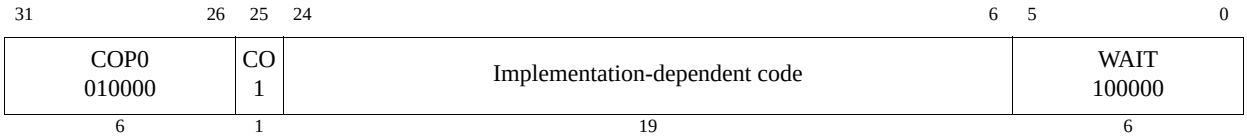
```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Invalid Operation, Unimplemented Operation



Format: `WAIT` **MIPS32**

Purpose: Enter Standby Mode

Wait for Event

Description:

The WAIT instruction performs an implementation-dependent operation, usually involving a lower power mode. Software may use the code bits 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 code bits 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:

Prior to MIPS32 Release 6: The operation of the processor is **UNDEFINED** if an WAIT is executed in the delay slot of a branch or jump instruction.

MIPS32 Release 6 implementations are required to signal a Reserved Instruction Exception if WAIT is encountered in the delay slot or forbidden slot of a branch or jump instruction.

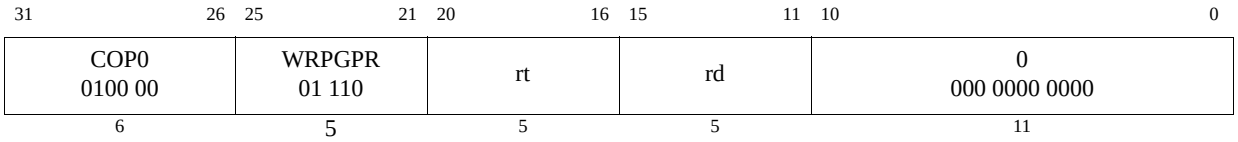
If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

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: Write to GPR in Previous Shadow Set

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.

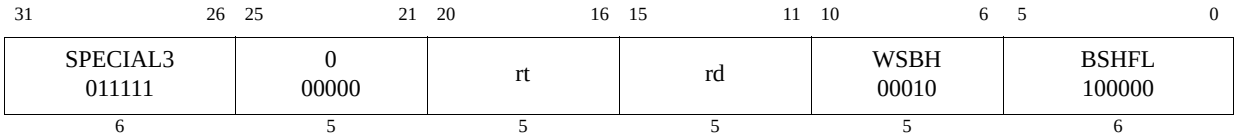
Operation:

```
SGPR[SRSCtlPSS, rd] ← GPR[rt]
```

Exceptions:

Coprocessor Unusable

Reserved Instruction



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

Purpose: Word Swap Bytes Within Halfwords

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 *r* 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[r]23..16 || GPR[r]31..24 || GPR[r]7..0 || GPR[r]15..8)
```

Exceptions:

Reserved Instruction

Programming Notes:

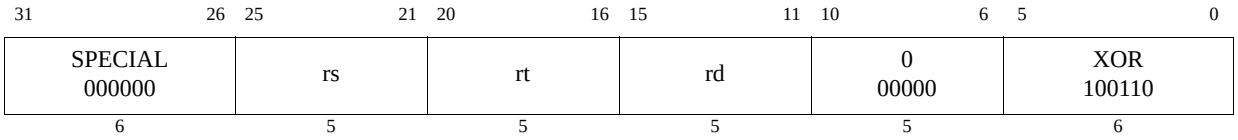
The WSBH instruction can be used to convert halfword and word data of one endianness to another endianness. The endianness of a word value can be converted using the following sequence:

```
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.



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

Purpose: Exclusive OR

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:

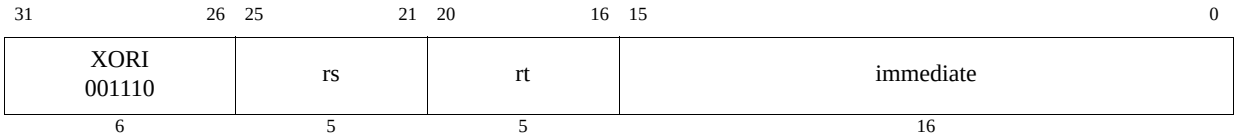
None

Operation:

```
GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None



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

Purpose: Exclusive OR Immediate

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

Appendix A

Instruction Bit Encodings

A.1 Instruction Encodings and Instruction Classes

Instruction encodings are presented in this section; field names are printed here and throughout the book in *italics*.

When encoding an instruction, the primary *opcode* field is encoded first. Most *opcode* values completely specify an instruction that has an *immediate* value or offset.

Opcode values that do not specify an instruction instead specify an instruction class. Instructions within a class are further specified by values in other fields. For instance, *opcode* REGIMM specifies the *immediate* instruction class, which includes conditional branch and trap *immediate* instructions.

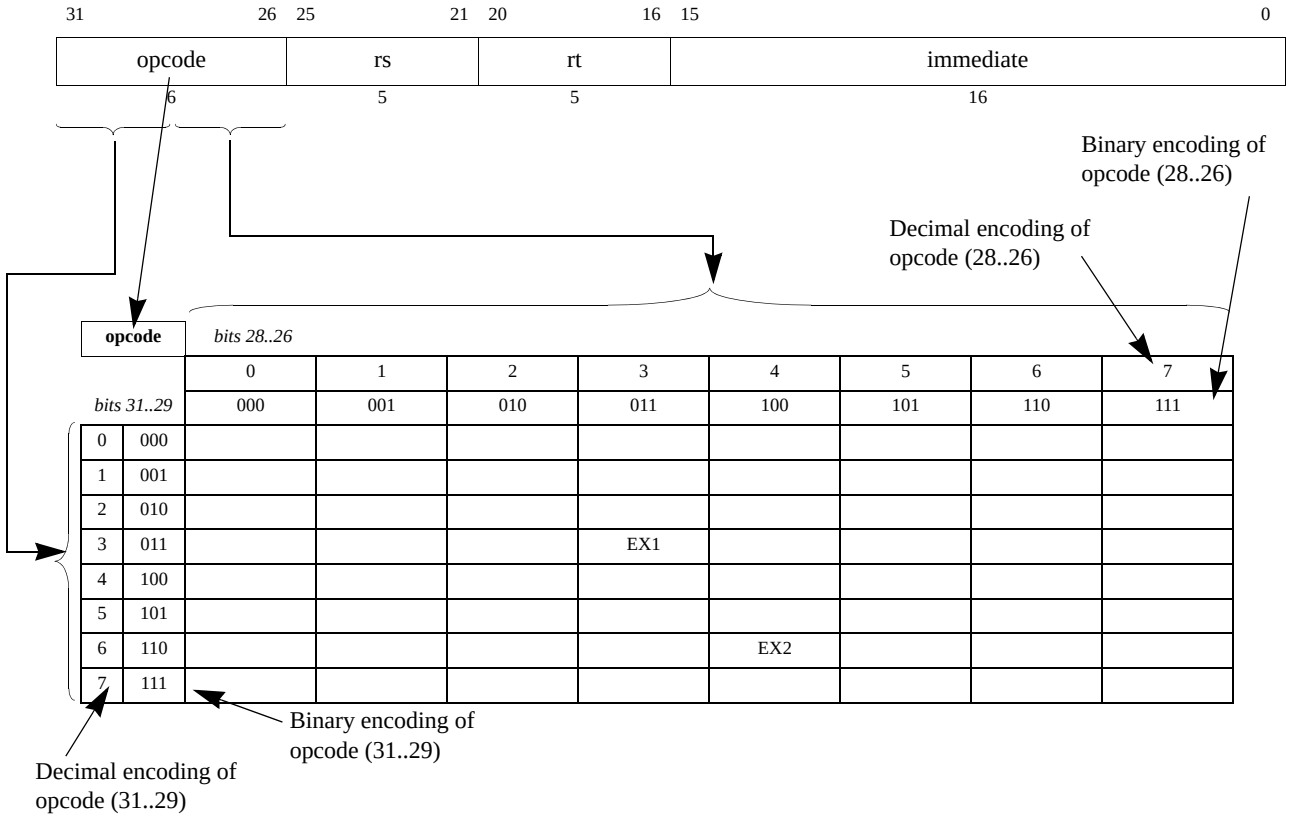
A.2 Instruction Bit Encoding Tables

This section provides various bit encoding tables for the instructions of the MIPS64® ISA.

Figure A.1 shows a sample encoding table and the instruction *opcode* field this table encodes. Bits 31..29 of the *opcode* field are listed in the leftmost columns of the table. Bits 28..26 of the *opcode* field are listed along the topmost rows of the table. Both decimal and binary values are given, with the first three bits designating the row, and the last three bits designating the column.

An instruction's encoding is found at the intersection of a row (bits 31..29) and column (bits 28..26) value. For instance, the *opcode* value for the instruction labeled EX1 is 33 (decimal, row and column), or 011011 (binary). Similarly, the *opcode* value for EX2 is 64 (decimal), or 110100 (binary).

Figure A.1 Sample Bit Encoding Table



Tables A.2 through A.23 describe the encoding used for the MIPS64 ISA. Table A.1 describes the meaning of the symbols used in the tables.

Table A.1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction must cause a Reserved Instruction Exception.
δ	(Also *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
β	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level or a new revision of the Architecture. Executing such an instruction must cause a Reserved Instruction Exception.
⊥	Operation or field codes marked with this symbol represent instructions which are not legal if the processor is configured to be backward compatible with MIPS32 processors. If the processor is executing with 64-bit operations enabled, execution proceeds normally. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).

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

Symbol	Meaning
∇	Operation or field codes marked with this symbol represent instructions which were only legal if 64-bit operations were enabled on implementations of Release 1 of the Architecture. In Release 2 of the architecture, operation or field codes marked with this symbol represent instructions which are legal if 64-bit floating point operations are enabled. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
Δ	Instructions formerly marked ∇ in some earlier versions of manuals, corrected and marked Δ in revision 5.03. Legal on MIPS64r1 but not MIPS32r1; in release 2 and above, legal in both MIPS64 and MIPS32, in particular even when running in “32-bit FPU Register File mode”, FR=0, as well as FR=1.
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).
θ*	MIPS32 Release 6 reserves the SPECIAL2 encodings. pre-MIPS32 Release 2 the SPECIAL2 encodings were available for customer use as UDIs. Otherwise like θ above.
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 optional Module or Application Specific Extensions. If the Module/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.
6N	Instruction added by MIPS32 Release 6. “N” for “new”.
6Nm	New MIPS32 Release 6 encoding for a pre-MIPS32 Release 6 instruction that has been moved. “Nm” for “New (moved)

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

Symbol	Meaning	
6Rm	pre-MIPS Release 6 instruction encoding moved by MIPS32 Release 6. "Removed" for "(moved elsewhere)".	6Rm and 6R instructions signal a Reserved Instruction Exception when executed by a MIPS32 Release 6 implementation.
6R	pre-MIPS32 Release 6 instruction encoding removed by MIPS32 Release 6. "R" for "removed".	

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 *see B*C*[6N] *family*[1]δ	BGTZ *see B*C*[6N] *family*[1]δ
1	001	ADDI[6R] *see B*C*[6N] *family*[1]δ	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI[2] / AUI[6N]
2	010	*COP0* δ	*COP1* δ	*COP2* θδ	*COP1X* δ[6R]	BEQL[6R] ϕ	BNEL[6R] ϕ	BLEZL[6R] ϕ *see B*C*[6N] *family*[1]δ	BGTZL[6R] ϕ *see B*C*[6N] *family*[1]δ
3	011	DADDI[6R]⊥ *see B*C*[6N] *family*[1]δ	DADDIU ⊥	LDL[6R] ⊥	LDR[6R] ⊥	*SPECIAL2* [6R] δ θ∗	JALX ε[6R] DAUI[6N]	*MSA* εδ	*SPECIAL3*[3] δ⊕
4	100	LB	LH	LWL[6R]	LW	LBU	LHU	LWR[6R]	LWU ⊥
5	101	SB	SH	SWL[6R]	SW	SDL[6R] ⊥	SDR[6R] ⊥	SWR[6R]	CACHE[6Rm]
6	110	LL[6Rm]	LWC1	LWC2[6Rm] θ *BC*[6N] *see B*C*[6N] *family*[1]δ	PREF[6Rm]	LLD[6Rm] ⊥	LDC1	LDC2[6Rm] θ *BEQZC/JIC*[6N] *see B*C*[6N] *family*[1]δ	LD ⊥
7	111	SC[6Rm]	SWC1	SWC2[6Rm] θ *BALC*[6N] *see B*C*[6N] *family*[1]δ	*PC-rel*[6N] *family*	SCD[6Rm] ⊥	SDC1	SDC2[6Rm] θ *BNEZC/JIALC*[6N] *see B*C*[6N] *family*[1]δ	SD ⊥

1. See Table A.28 for the encodings of the compact control transfers, marked "*see B*C*[6N] *family*[1]δ" in the table above. Where only one or two MIPS32 Release 6 compact CTIs occupy a primary opcode they are indicated in the table; where too many to fit they are not indicated. In both of these cases, the "see B*C" marking is applied. In some cases these conflict with the pre-MIPS32 Release 6 instructions occupying the same cell, marked 6R for removed; where not marked 6R, e.g. BLEZ, BGTZ, further fields disambiguate as described in Table A.28.
2. Prior to MIPS32 Release 6 instruction LUI is a special case of MIPS32 Release 6 instruction AUI.
3. 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* [6R]δ	*SRL* δ	SRA	SLLV	LSA[6N] ε	*SRLV* δ	SRAV
1	001	JR[2,3,6R]	JALR[2]	MOVZ[6R]	MOVN[6R]	SYSCALL	BREAK	SDBBP[6Rm]	SYNC
2	010	MFHI[6R] CLZ[6Nm]	MTHI[6R] CLO[6Nm]	MFLO[6R] DCLO[6Nm]	MTLO[6R] DCLO[6Nm]	DSLLV ⊥	DLSA[6N] ε⊥	DSRLV δ⊥	DSRAV ⊥
3	011[4]	[4]MULT[6R] MUL/MUH[6N]	[4]MULTU[6R] MULU/MUHU[6N]	[4]DIV[6R] DIV/MOD[6N]	[4]DIVU[6R] DIVU/MODU[6N]	[4]DMULT ⊥ DMUL/DMUH[6N]	[4]DMULTU ⊥ DMULU/DMUHU[6N]	[4]DDIV ⊥ DDIV/DMOD[6N]	[4]DDIVU ⊥ DDIVU/DMODU[6N]
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	SELEQZ[6N]	TNE	SELNEZ[6N]
7	111	DSLL ⊥	*	*DSRL* δ⊥	DSRA ⊥	DSLL32 ⊥	*	*DSRL32* δ⊥	DSRA32 ⊥

1. Specific encodings of the *rt*, *rd*, and *sa* fields are used to distinguish among the SLL, NOP, SSNOP, EHB and PAUSE functions. MIPS32 Release 6 makes SSNOP equivalent to NOP.
2. Specific encodings of the *hint* field are used to distinguish JR from JR.HB and JALR from JALR.HB
3. MIPS32 Release 6 removes JR and JR.HB. JALR with rd=0 provides functionality equivalent to JR. JALR.HB with rd=0 provides functionality equivalent to JR.HB. Assemblers should produce the new instruction when encountering the old mnemonic.
4. Specific encodings of the *sa* field are used to distinguish pre-MIPS32 Release 6 and MIPS32 Release 6 integer multiply and divide instructions. See Table A.25 on page 764, which shows that the encodings do not conflict. The pre-MIPS32 Release 6 divide instructions signal Reserved Instruction exception on MIPS32 Release 6. Note that the same mnemonics are used for pre-MIPS32 Release 6 divide instructions that return both quotient and remainder, and MIPS32 Release 6 divide instructions that return only quotient, with separate MOD instructions for the remainder.

DocumentMarginNote

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[6R] ϕ	BGEZL[6R] ϕ	*	*	DAHI[6N]	ε
1	01	TGEI[6R]	TGEIU[6R]	TLTI[6R]	TLTIU[6R]	TEQI[6R]	*	TNEI[6R]	*
2	10	BLTZAL[6R] NAL[6N]	BGEZAL[6R] BAL[6N]	BLTZALL[6R] ϕ	BGEZALL[6R] ϕ	*	*	*	*
3	11	*	*	*	*	ε	ε	DATI[6N]	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[6R] θ*	MADDU[6R] θ*	MUL[6R] θ*	θ*	MSUB[6R] θ*	MSUBU[6R] θ*	θ*	θ*
1	001	ε 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[6Rm] θ*	CLO[6Rm] θ*	θ*	θ*	DCLZ⊥ θ*	DCLO⊥ θ*	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[6Rm] σ θ*

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	ε	ε	ε	*	ε	ε	*	*
2	010	ε	ε	ε	ε	ε	ε	ε	ε
3	011	ε	LWLE[6R]	LWRE[6R]	CACHEE	SBE	SHE	SCE	SWE
4	100	BSHFL ⊕δ	SWLE[6R]	SWRE[6R]	PREFE	*DBSHFL* ⊥⊕δ	CACHE[6Nm]	SC[6Nm]	SCD[6Nm]
5	101	LBUE	LHUE	*	*	LBE	LHE	LLE	LWE
6	110	ε	ε	*	*	ε	PREF[6Nm]	LL[6Nm]	LLD[6Nm]
7	111	ε	*	*	RDHWR ⊕	ε	*	*	*

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*[6R][1] Encoding of *tf* Bit

tf	*bit 16*	
	0	1
	MOVF[6R]	MOVT[6R]

1. MIPS32 Release 6 removes the MOVCI instruction family (MOVT and MOVF).

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	BITSWAP[6N] (BSHFL)		WSBH (BSHFL) DSBH (DBSHFL)			DSHD (DBSHFL)		
1	01	ALIGN[6N] (BSHFL) DALIGN[6N] (DBSHFL)							
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 ⊥	*	ε	MTC0	DMTC0 ⊥	*	*
1	01	ε	*	RDPGPR ⊕	*MFMC0*[1] δ⊕	ε	*	WRPGPR ⊕	*
2	10	C0 δ							
3	11								

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

Table A.15 MIPS64 *COP0* Encoding of Function Field When *rs=CO*

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	TLBR	TLBWI	TLBINV	TLBINVF	*	TLBWR	*
1	001	TLBP	ε	ε	ε	ε	*	ε	*
2	010	ε	*	*	*	*	*	*	*
3	011	ERET	*	*	*	*	*	*	DERET σ
4	100	WAIT	*	*	*	*	*	*	*
5	101	ε	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	ε	*	*	*	*	*	*	*

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[6R] δ	BC1ANY2[6R] δεV BC1EQZ[6N]	BC1ANY4[6R] δεV	BZ.V ε	*	BC1NEZ[6N]	*	BNZ.V ε
2	10	S δ	D δ	*	*	W δ	L δ	PS[6R] δ	*
3	11	BZ.B ε	BZ.H ε	BZ.W ε	BZ.D ε	BNZ.B ε	BNZ.H ε	BNZ.W ε	BNZ.D ε

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.S	SUB.S	MUL.S	DIV.S	SQRT.S	ABS.S	MOV.S	NEG.S
1	001	ROUND.L.S V	TRUNC.L.S V	CEIL.L.S V	FLOOR.L.S V	ROUND.W.S	TRUNC.W.S	CEIL.W.S	FLOOR.W.S
2	010	SEL.S[6N]	MOVCF.S[6R] δ	MOVZ.S[6R]	MOVN.S[6R]	SELEQZ.S[6N]	RECIP.S Δ	RSQRT.S Δ	SELNEZ.S[6N]
3	011	MADDF.S[6N]	MSUBF.S[6N]*	RINT.S[6N]	CLASS.S[6N]	RECIP2.S εV MIN.S[6N]	RECIP1.S ε MAX.S[6N]	RSQRT1.S εV MINA.S[6N]	RSQRT2.S ε MAXA.S[6N]
4	100	*	CVT.D.S	*	*	CVT.W.S	CVT.L.S V	CVT.PS.S[6R]V	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F.S[6R] CABS.F.S εV	C.UN.S[6R] CABS.UN.S εV	C.EQ.S[6R] CABS.EQ.S εV	C.UEQ.S[6R] CABS.UEQ.S εV	C.OLT.S[6R] CABS.OLT.S εV	C.ULT.S[6R] CABS.ULT.S εV	C.OLE.S[6R] CABS.OLE.S εV	C.ULE.S[6R] CABS.ULE.S εV
7	111	C.SF.S[6R] CABS.SF.S εV	C.NGLE.S[6R] CABS.NGLE.S εV	C.SEQ.S[6R] CABS.SEQ.S εV	C.NGL.S[6R] CABS.NGL.S εV	C.LT.S[6R] CABS.LT.S εV	C.NGE.S[6R] CABS.NGE.S εV	C.LE.S[6R] CABS.LE.S εV	C.NGT.S[6R] CABS.NGT.S εV

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.D	SUB.D	MUL.D	DIV.D	SQRT.D	ABS.D	MOV.D	NEG.D
1	001	ROUND.L.D ∇	TRUNC.L.D ∇	CEIL.L.D ∇	FLOOR.L.D ∇	ROUND.W.D	TRUNC.W.D	CEIL.W.D	FLOOR.W.D
2	010	SEL.D^{6N}	MOVCF.D^{6R} δ	MOVZ.D^{6R}	MOVN.D^{6R}	SELEQZ.D^{6N}	RECIP.D Δ	RSQRT.D Δ	SELNEZ.D^{6N}*
3	011	MADDF.D^{6N}	MSUBF.D^{6N}*	RINT.D^{6N}	CLASS.D^{6N}	RECIP2.D ε∇ MIN.D^{6N}	RECIP1.D ε MAX.D^{6N}	RSQRT1.D ε∇ MINA.D^{6N}	RSQRT2.D ε MAXA.D^{6N}
4	100	CVT.S.D	*	*	*	CVT.W.D	CVT.L.D ∇	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F.D^{6R} CABS.F.D ε∇	C.UN.D^{6R} CABS.UN.D ε∇	C.EQ.D^{6R} CABS.EQ.D ε∇	C.UEQ.D^{6R} CABS.UEQ.D ε∇	C.OLT.D^{6R} CABS.OLT.D ε∇	C.ULT.D^{6R} CABS.ULT.D ε∇	C.OLE.D^{6R} CABS.OLE.D ε∇	C.ULE.D^{6R} CABS.ULE.D ε∇
7	111	C.SF.D^{6R} CABS.SF.D ε∇	C.NGLE.D^{6R} CABS.NGLE.D ε∇	C.SEQ.D^{6R} CABS.SEQ.D ε∇	C.NGL.D^{6R} CABS.NGL.D ε∇	C.LT.D^{6R} CABS.LT.D ε∇	C.NGE.D^{6R} CABS.NGE.D ε∇	C.LE.D^{6R} CABS.LE.D ε∇	C.NGT.D^{6R} CABS.NGT.D ε∇

Table A.19 MIPS64 *COP1* Encoding of Function Field When *rs=W* or *L*[1] [2]

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	CMP.AF.S/D^{6N}	CMP.UN.S/D^{6N}	CMP.EQ.S/D^{6N}	CMP.UEQ.S/D^{6N}	CMP.OLT.S/D^{6N}	CMP.ULT.S/D^{6N}	CMP.OLE.S/D^{6N}	CMP.ULE.S/D^{6N}
1	001	CMP.SAF.S/D^{6N}	CMP.SUB.S/D^{6N}	CMP.SEQ.S/D^{6N}	CMP.SUEQ.S/D^{6N}	CMP.SLT.S/D^{6N}	CMP.SULT.S/D^{6N}	CMP.SLE.S/D^{6N}	CMP.SULE.S/D^{6N}
2	010	*	CMP.OR.S/D^{6N}	CMP.UNE.S/D^{6N}	CMP.NE.S/D^{6N}	*	*	*	*
3	011	*	CMP.SOR.S/D^{6N}	CMP.SUNE.S/D^{6N}	CMP.SNE.S/D^{6N}	*	*	*	*
4	100	CVT.S.W/L	CVT.D.W/L	*	*	*	*	CVT.PS.PW^{6R}ε∇	*
5	101	*	*	*	*	*	*	*	*
6	110								
7	111								

1. Format type *L* is legal only if 64-bit floating point operations are enabled.
2. MIPS32 Release 6 introduces the CMP.condn.fmt instruction family, where .fmt=S or D, 32 or 64 bit floating point. However, .S and .D for CMP.condn.fmt are encoded as .W 10100 and .L 10101 in the “standard” format. The conditions tested are encoded the same way for pre-MIPS32 Release 6 C.cond.fmt and MIPS32 Release 6 CMP.cond.fmt, except that MIPS32 Release 6 adds new conditions not present in C.cond.fmt. MIPS32 Release 6, however, has changed the recommended mnemonics for the CMP.condn.fmt to be consistent with the IEEE standard rather than pre-MIPS32 Release 6. See the table in the description of CMP.cond.fmt in Volume II of the MIPS Architecture Reference Manual,which shows the correspondence between pre-MIPS32 Release 6 C.cond.fmt, MIPS32 Release 6 CMP.cond.fmt, and MSA FC*.fmt / FS*.fmt floating point comparisons.

Table A.20 MIPS64 *COP1* Encoding of Function Field When *rs=PS*[1][6R][2]

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

1. Format type *PS* is legal only if 64-bit floating point operations are enabled.
2. MIPS32 Release 6 removes format type PS (paired single). MSA (MIPS SIMD Architecture) may be used instead.

Table A.21 MIPS64 *COP1* Encoding of *tf* Bit When *rs=S*, D, or PS[6R], Function=*MOVCF*[6R][1]

tf	bit 16	
	0	1
	MOVF.fmt[6R]	MOVT.fmt[6R]

1. MIPS32 Release 6 removes the MOVCF instruction family (MOVF.fmt and MOVT.fmt), replacing them by SEL.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[6R] θ	BC2EQZ[6N]	LWC2[6Rm]θ	SWC2[6Rm]θ	θ	BC2NEZ[6N]θ	LDC2[6Rm]θ	SDC2[6Rm]θ
2	10	*C2* θ δ							
3	11								

Table A.23 MIPS64 *COP1X*[6R][1] 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	LWXC1[6R] Δ	LDXC1[6R] Δ	*	*	*	LUXC1[6R] ∇	*	*
1	001	SWXC1[6R] Δ	SDXC1[6R] Δ	*	*	*	SUXC1[6R] ∇	*	PREFX[6R] Δ
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	ALNV.PS[6R]∇	*
4	100	MADD.S[6R]Δ[2]	MADD.D[6R]Δ[2]	*	*	*	*	MADD.PS[6R]∇	*
5	101	MSUB.S[6R]Δ[2]	MSUB.D[6R]Δ[2]	*	*	*	*	MSUB.PS[6R]∇	*
6	110	NMADD.S[6R]Δ[2]	NMADD.D[6R]Δ[2]	*	*	*	*	NMADD.PS[6R]∇	*
7	111	NMSUB.S[6R]Δ[2]	NMSUB.D[6R]Δ[2]	*	*	*	*	NMSUB.PS[6R]∇	*

1. MIPS32 Release 6 removes format type PS (paired single). MSA (MIPS SIMD Architecture) may be used instead.
2. MIPS32 Release 6 removes all pre-MIPS32 Release 6 COP1X instructions, of the form 010011 - COP1X.PS, non-fused FP multiply adds, and indexed and unaligned loads, stores, and prefetches.

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 ranging from Table A.16 to 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)**		**Mnemonic**	**Name**	**Bit Width**	**Data Type**
Decimal	**Hex**	**Decimal**	**Hex**				
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
				See note below: **MIPS32 Release 6 CMP.condn.S/D encoded as W/L.**			
17	11	1	1	D	Double	64	Floating Point
				See note below: **MIPS32 Release 6 CMP.condn.S/D encoded as W/L.**			
18..19	12..13	2..3	2..3	Reserved for future use by the architecture.			
20	14	4	4	W	Word	32	Fixed Point
				See note below: **MIPS32 Release 6 CMP.condn.S/D encoded as W/L.**			
21	15	5	5	L	Long	64	Fixed Point
				See note below: **MIPS32 Release 6 CMP.condn.S/D encoded as W/L.**			

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)		Mnemonic	Name	Bit Width	Data Type
Decimal	**Hex**	**Decimal**	**Hex**				
22	16	6	6	PS	Paired Single	2 × 32	Floating Point
				MIPS32 Release 6 removes the PS format, and reserves it for future use			
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.			

Note: MIPS32 Release 6 CMP.condn.S/D encoded as W/L: as described in Table A.19 on page 759, “MIPS64 COP1 Encoding of Function Field When rs=W or L” on page 759, MIPS32 Release 6 uses certain instruction encodings with the *rs* (*fmt*) field equal to 11000 (W) or 11001 (L) to represent S and D respectively, for the instruction family CMP.condn.fmt.

A.4 MIPS32 Release 6 Instruction Encodings

MIPS32 Release 6 adds several new instructions, removes several old instructions, and changes the encodings of several pre-MIPS32 Release 6 instructions. In many cases, the old encodings for instructions moved or removed are required to signal the Reserved Instruction on MIPS32 Release 6, so that uses of old instructions can be trapped, and emulated or warned about; but in several cases the old encodings have been reused for new MIPS32 Release 6 instructions.

These instruction encoding changes are indicated in the tables above. MIPS32 Release 6 new instructions are superscripted 6N; MIPS32 Release 6 removed instructions are superscripted 6R; MIPS32 Release 6 instructions that have been moved are marked 6Rm at the pre-MIPS32 Release 6 encoding that they are moved from, and 6Nm at the new MIPS32 Release 6 encoding that it is moved to. Encoding table cells that contain both a non-MIPS32 Release 6 instruction and a MIPS32 Release 6 instruction superscripted 6N or 6Nm indicate a possible conflict, although in many cases footnotes indicate that other fields allow the distinction to be made.

The tables below show the further decoding in MIPS32 Release 6 for field classes (instruction encoding families) indicated in other tables.

Instruction encodings are also illustrated in the instruction descriptions in Volume II. Those encodings are authoritative. The instruction encoding tables in this section, above, based on bitfields, are illustrative, since they cannot completely indicate the new tighter encodings.

MUL/DIV family encodings: Table A.25 below shows the MIPS32 Release 6 integer family of multiply and divide instructions encodings, as well as the pre-MIPS32 Release 6 instructions they replace. The MIPS32 Release 6 and pre-MIPS32 Release 6 instructions share the same primary opcode, bits 31-26 = 000000, and share the function code, bits 5-0, with their pre-MIPS32 Release 6 counterparts, but are distinguished by bits 10-6 of the instruction. The pre-MIPS32 Release 6 instructions signal a Reserved Instruction exception on MIPS32 Release 6 implementations.

DocumentMarginNote

However, the instruction names collide: pre-MIPS32 Release 6 MIPS32 Release 6 DIV, DIVU, DDIV, DDIVU are actually distinct instructions, although they share the same mnemonics. The pre-MIPS32 Release 6 instructions produce two results, both quotient and remainder in the HI/LO register pair, while the MIPS32 Release 6 DIV instruction produce only a single result, the quotient. It is possible to distinguish the conflicting instructions in assembly by looking at how many register operands the instructions have, two versus three.

As of MIPS32 Release 6, all of pre-MIPS32 Release 6 instruction encodings that are removed are required to signal the reserved instruction exception, as are all in the vicinity 000000.xxxxx.xxxxx.aaaaa.011xxx, i.e. all with the primary opcodes and function codes listed in Table A.25, with the exception of the aaaaa field values 00010 and 00011 for the new instructions.

Table A.25 MIPS32 Release 6 MUL/DIV encodings

pre-MIPS32 Release 6 removed ~~struck-through~~

`00000.rs.rt.rd.aaaaa.function6`

function bits 5-0	aaaaa, bits 10-6		
	00000 and rd = 00000 (bits 15-11)	**00010**	**00011**
011 **000**	~~MULT~~[6R]	MUL[6N]	MUH[6N]
011 **001**	~~MULTU~~[6R]	MULU[6N]	MUHU[6N]
011 **010**	~~DIV~~[6R]	DIV[6N]	MOD[6N]
011 **011**	~~DIVU~~[6R]	DIVU[6N]	MODU[6N]
011 **100**	~~DMULT~~[6R]	DMUL[6N]	DMUH[6N]
011 **101**	~~DMULTU~~[6R]	DMULU[6N]	MUHU[6N]
011 **110**	~~DDIV~~[6R]	DDIV[6N]	DMOD[6N]
011 **111**	~~DDIVU~~[6R]	DDIVU[6N]	DMODU[6N]

PC-relative family encodings: Table A.26 and Table A.27 present the PC-relative family of instruction encodings. Table A.26 in traditional form, Table A.27 in the bitstring form that clearly shows the immediate varying from 19 bits to 16 bits.

DocumentMarginNote

Table A.26 MIPS32 Release 6 PC-relative family encoding

`111011.rs.TTTTT.imm16`

rs		*bits 18-16*							
		0	1	2	3	4	5	6	7
bits 20-19		000	001	010	011	100	101	110	111
0	00	ADDIUP[6N] imm19							
1	01	LWP[6N] imm19							
2	10	LWUP[6N] imm19							
3	11	LDP[6N] imm18				reserved (RI)		AUIP[6N] imm16	ALUIP[6N] imm16

Table A.27 MIPS32 Release 6 PC-relative family encoding bitstrings

111011.rs.*

encoding	instruction
111011.rs.**00**.<-----imm19>	ADDIUP[6N]
111011.rs.**01**.<----disp19>	LWP[6N]
111011.rs.**10**.<----disp19>	LWUP[6N]
111011.rs.**110**.<---disp18>	LDP[6N]
111011.rs.**1110**.<---imm17>	reserved, signal RI[6N]
111011.rs.**11110**.<--imm16>	AUIP[6N]
111011.rs.**11111**.<--imm16>	ALUIP[6N]

B*C compact branch and jump encodings: In several cases MIPS32 Release 6 uses much tighter instruction encodings than previous releases of the MIPS architecture, reducing redundancy, to allow more instructions to be encoded. Instead of purely looking at bitfields, MIPS32 Release 6 defines encodings that compare different bitfields: e.g. the encoding 010110.rs.rt.offset16 is BGEC if neither rs nor rt are 00000 and rs is not equal to rt, but is BGEZC if rs is the same as rt, and is BLEZC if rs is 00000 and rt is not. (The encoding with rt 00000 and arbitrary rs is the pre-MIPS32 Release 6 instruction BLEZL.rs.00000.offset16, a branch likely instruction which is removed by MIPS32 Release 6, and whose encoding is required to signal the Reserved Instruction exception.)

This tight instruction encoding motivates the bitstring and constraints notation for MIPS32 Release 6 instruction encodings

```
BLEZC rt      010110.00000.rt.offset16, rt!=0
BGEZC rt      010110.rs=rt.rt.offset16, rs!=0, rt!=0, rs=rt
BGEC rs,rt    010110.rs.rt.offset16,    rs!=0, rt!=0, rs!=rt
BLEZL rt      010110.00000.rt.offset16, rs=0
```

and the equivalent constraints indicated in the instruction encoding diagrams for the instruction descriptions in Volume II. Table A.28 below shows the B*C compact branch encodings, which use constraints such as RS = RT. pre-MIPS32 Release 6 encodings that are removed by MIPS32 Release 6 are shaded darkly, while the remaining redundant encodings are shaded lightly or stippled.

Note that the pre-MIPS32 Release 6 instructions BLEZL, BGTZL, BLEZ, and BGTZ do not conflict with the new MIPS32 Release 6 instructions they are tightly packed with in the encoding tables, but the ADDI, DADDI, LWC2, SWC2, LDC2 and SDC2 truly conflict.

Table A.28 B*C compact branch encodings

Primary Opcode	Constraints involving rs and rt fields			
	$rs/rt_{0/NZ}$		NZ_{rs} =/</> NZ_{rt}	
010 110	$0_{rs}\ 0_{rt}$	~~BLEZL~~[6R]	BGEZC[6N]	=
	$0_{rs}\ NZ_{rt}$	BLEZC[6N]	BGEC[6N] (BLEC)	<
	$NZ_{rs}\ 0_{rt}$	~~BLEZL~~[6R]		>
				$rs_{NZ} \neq rt_{NZ}$
000 110	$0_{rs}\ 0_{rt}$	~~BLEZ~~[6R]	BGEZALC[6N]	=
	$0_{rs}\ NZ_{rt}$	BLEZALC[6N]	BGEUC[6N] (BLEUC)	<
	$NZ_{rs}\ 0_{rt}$	~~BLEZ~~[6R]		>
				$rs_{NZ} \neq rt_{NZ}$
010 111	$0_{rs}\ 0_{rt}$	~~useless BGTZL~~[6R]	BLTZC[6N]	=
	$0_{rs}\ NZ_{rt}$	BGTZC[6N]	BLTC[6N] (BGTC)	<
	$NZ_{rs}\ 0_{rt}$	~~BGTZL~~[6R]		>
				$rs_{NZ} \neq rt_{NZ}$
000 111	$0_{rs}\ 0_{rt}$	~~useless BGTZ~~[6R]	BLTZALC[6N]	=
	$0_{rs}\ NZ_{rt}$	BGTZALC[6N]	BLTUC[6N] (BGTUC)	<
	$NZ_{rs}\ 0_{rt}$	~~BGTZ~~[6R]		>
				$rs_{NZ} \neq rt_{NZ}$
001 000	~~ADDI~~			
	$0_{rs}\ NZ_{rt}$	BEQZALC[6N]	BEQC[6N]	<
	$0_{rs}\ 0_{rt}$	useless BEQZALC		=
	$NZ_{rs}\ 0_{rt}$		BOVC[6N]	>
				$rs_{NZ} \geq rt_{0,NZ}$
011 000	~~DADDI~~[6R]			
	$0_{rs}\ NZ_{rt}$	BNEZALC[6N]	BNEC[6N]	<
	$0_{rs}\ 0_{rt}$	useless BNEZALC		=
	$NZ_{rs}\ 0_{rt}$		BNVC[6N]	>
				$rs_{NZ} \geq rt_{0,NZ}$
110 110	~~LDC2~~[6R]			
	$0_{rs}\ 0/NZ_{rt}$ / $0_{rs}\ NZ_{rt}$	JIC[6N] rt+disp16	BEQZC[6N] rs_{NZ}, ofs21	<
	$0_{rs}\ 0_{rt}$			=
	$NZ_{rs}\ 0_{rt}$			>
				$NZ_{rs}\ 0/NZ_{rt}$
110 110	~~SDC2~~[6R]			
	$0_{rs}\ 0/NZ_{rt}$ / $0_{rs}\ NZ_{rt}$	JIALC[6N] rt+disp16	BNEZC[6N] rs_{NZ}, ofs21	<
	$0_{rs}\ 0_{rt}$			=
	$NZ_{rs}\ 0_{rt}$			>
				$NZ_{rs}\ 0/NZ_{rt}$
110 010	~~LWC2~~[6R]			
		BC[6N] ofs26<<2		
				$0/NZ_{rs}\ 0/NZ_{rt}$
111 010	~~SWC2~~[6R]			
		BALC[6N] ofs26<<2		
				$0/NZ_{rs}\ 0/NZ_{rt}$

DocumentMarginNote

Appendix B

Misaligned Memory Accesses

Prior to Release 5 the MIPS architectures[1] require “natural” alignment of memory operands for most memory operations. Instructions such as LWL and LWR are provided so that unaligned accesses can be performed via instruction sequences. As of Release 5 of the Architecture the MSA (MIPS SIMD Architecture) supports 128 bit vector memory accesses, and does NOT require these MSA vector load and store instructions to be naturally aligned, i.e. it permits misaligned MSA vector loads and stores..

MIPS32 Release 6 requires misaligned memory access support for all ordinary memory access instructions (e.g. LW/SW, LWC1/SWC1). Misaligned support is not provided for certain special memory accesses such as atomics (e.g. LL/SC). Each instruction description in Volume II of the Architecture Reference Manual indicates whether misalignment support is provided. MIPS32 Release 6 removes the unaligned memory access instructions (e.g. LWL/LWR, LDL/LDR); these instructions are required to signal the Reserved Instruction exception. MIPS32 Release 6 also introduces the ALIGN instruction, which can also be used following a pair of LW instructions that the programmer has ensured are aligned, to emulate a misaligned load without using LWL/LWR, without the possible performance penalty of a misaligned access.

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

B.1 Terminology

This document uses the following terminology:

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

1. For example, see: **MIPS® Architecture For Programmers, Volume I-A: Introduction to the MIPS32® Architecture**, Document Number MD00082, Revision 3.50, September 20, 2012, `http://www.mips.com/auth/MD00082-2B-MIPS32INT-AFP-03.50.pdf`; sections 2.8.6.4 “Addressing Alignment Constraints” and 2.8.6.5 “Unaligned Loads and Stores” on page 38.

- Misaligned memory access instructions
  - More precisely: “instructions that permit, and are required to support, implicit misaligned memory accesses”.
  - Release 5 of the MIPS Architecture defines instructions, the MSA vector loads and stores, which may be aligned (e.g.128-bits on a 128 bit boundary), partially aligned (e.g. “element aligned”, see below), or misaligned. These may be called
  - MIPS32 Release 6 makes almost all memory access instructions into “instructions that permit implicit misaligned memory accesses” (with exceptions such as LL/SC).
- Aligned memory access instructions
  - More precisely: “instructions that do not permit, and which may be required to produce an exception, for misaligned addresses.
  - pre-MIPS32 Release 6, all MIPS memory access instructions, e.g. LW/SW, LL/SC, etc., required “natural alignment”, except for the “Unaligned Load and Store Instructions”, LWL/LWR and SWL/SWR.
  - MIPS32 Release 6 makes most ordinary memory access instructions (e.g LW/SW) require alignment.
  - MIPS32 Release 6 still requires alignment for a smaller set of memory access instructions, such as LL/SC.
- Misalignment is dynamic, known only when the address is computed (rather than static, explicit in the instruction as it is for LWL/LWR, etc.). We distinguish accesses for which the alignment is not yet known (“potentially misaligned”), from those whose alignment is known to be misaligned (“actually misaligned”), and from those for which the alignment is known to be naturally aligned (“actually aligned”).
  E.g. LL/SC instructions are never potentially misaligned., i.e. are always actually aligned (if they do not trap). MSA vector loads and stores are potentially misaligned, although the programmer or compiler may prevent actual misalignment.

B.2 Hardware versus software support for misaligned memory accesses

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

Systems that implement Release 5 or higher of the MIPS Architectures require support for misaligned memory operands for the following instructions:

- MSA (MIPS SIMD Architecture) vector loads and stores (128-bit quantities)

In Release 5 all misaligned memory accesses other than MSA continue to produce the Address Error Exception with the appropriate ErrorCode.

Systems that implement Release 6 or higher of the MIPS Architectures (MIPS32 Release 6) require support for misaligned memory operands for almost all memory access instructions, including:

- Byte loads and stores of course cannot be misaligned: LB, LBE, LBU, LBUE. Nor can cache PREF instructions, etc.

- CPU loads and stores: LH/SH, LHU/SHU, LW/SW, LWU/SWU, and MIPS64 LD/SD,
- The EVA versions of the above: LHE, LWE, MIPS64 LDE
- FPU loads and stores: LWC1/SWC1, LDC1/SDC1, LWXC1/SWXC1, LDXC1/SDXC1, LUXC1/SUXC1
- Coprocessor loads and stores: LWC2/SDC2, LDC2/SDC2.

These are collectively called the “Ordinary Memory Accesses”.

In particular, misalignment support is NOT provided for Nor is it provided for LL/SC. Nor for MIPS64 LDL/LDR and SDL/SDR, and LLD/SCD. Nor for the EVA versions LWLE/SWLE, LWRE/SWRE, LLE/SCE. All such instructions continue to produce the Address Error Exception if misaligned.

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

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

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

Trap (and Emulate) All Misaligneds:
E.g. it is permitted for a processor implementation to produce the Address Error Exception for all misaligned accesses. I.e. with the appropriate exception handler software,

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

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

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

Other mixes of hardware and software support are possible.

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

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

B.3 Detecting misaligned support

For MIPS32 Release 5 MSA misalignment support, it is sufficient to check that MSA is present, as defined by the appropriate reference manual[2]: i.e. support for misaligned MSA vector load and store instructions is required if the Config3 MSAP bit is set(CP0 Register 16, Select 3, bit 28).

For MIPS32 Release 6 misalignment support, it is sufficient to check the architecture revision level (COP0.Config.AR), as defined by the Privileged Resource Architecture.[3]

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

B.4 Misaligned semantics

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

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

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

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

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

E.g. uncorrectable ECC errors that occur halfway through a split store may violate single thread atomicity.

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

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

B.4.2 Permissions and misaligned memory accesses

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

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

2. E.g. MIPS® Architecture for Programmers, Volume IV-j: The MIPS32® SIMD Architecture Module, Document Number MD00866, 2013; or the corresponding documents for other MIPS Architectures such as MIPS64®
3. E.g. MIPS® Architecture Reference Manual, Volume III: The MIPS32® and microMIPS™ Privileged Resource Architecture, Document Number MD00088; or the corresponding documents for other MIPS Architectures such as MIPS64®.

This applies to all memory accesses, but in particular applies to misaligned split accesses, which can cross page boundaries and/or other boundaries that have different permissions. It *IS* permitted for a misaligned, in particular a page split memory access, to cross permission boundaries, as long as the access is permitted by permissions on both sides of the boundary. I.e. it is not required that the permissions be identical for all parts, just that all parts are permitted.

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

Architecture Rule B-6: If multiple exceptions arise for a given part of a misaligned memory access, then the same prioritization rules apply as for a non-misaligned memory access.

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

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

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

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

Architecture Rule B-10: The address reported by BadVaddr on page permission or TLB miss exceptions must be the lowest virtual address in the misaligned access for a page on which the exception is reported.

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

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

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

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

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

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

B.4.3 Misaligned Memory Accesses Past the End of Memory

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

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

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

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

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

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

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

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

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

B.4.4 TLBs and Misaligned Memory Accesses

A specific case of rules stated above:

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

Here “performed” the actual store, changing memory or cache data values, or the actual load, writing a destination register, or load side effects related to memory mapped I/O. It does not refer to microarchitectural side effects such as changing cache line state from M in another processor to S locally, nor to TLB state.

Note: this rules does NOT disallow emulating misaligned memory accesses via a trap handler that performs the access a byte at a time, even though a TLB miss may occur for a later byte after an earlier byte has been written. Such

a trap handler is emulating the entire misaligned. A TLB miss in the emulation code will return to the emulation code, not to the original misaligned memory instruction.

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

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

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

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

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

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

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

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

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

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

B.4.5 Memory Types and Misaligned Memory Accesses

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

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

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

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

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

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

B.4.6 Misaligneds, Memory Ordering, and Coherence

This section discusses single and multithread atomicity and multithread memory ordering for misaligned memory accesses. But the overall Misaligned Memory Accesses specification, does not address issues for potentially but not actually misaligned memory references. Documents such as the MIPS Coherence Protocol Specification define such behavior.[4]

B.4.6.1 Misaligneds are Single Thread Atomic

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

Architecture Rule B-1: “Misaligned memory accesses are atomic with respect to a single thread (with limited exceptions noted in other rules).” on page 771.

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

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

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

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

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

Machine Check Exceptions (a) are implementation dependent, (b) could potentially include a wide number of

4. E.g. MIPS Coherence Protocol Specification (AFP Version), Document Number MD00605, Revision 0.100. June 25, 2008. Updates and revisions of this document are pending.

processor internal inconsistencies. However, at the time of writing the only Machine Check Exceptions that are defined are (a) detection of multiple matching entries in the TLB, and (b) inconsistencies in memory data structures encountered by the hardware page walker page table. Neither of these should cause a violation of single thread atomicity for misaligneds. In general, no errors related to virtual memory addresses should cause violations of single thread atomicity.

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

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

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

B.4.6.2 Misaligneds are not Multiprocessor/Multithread Atomic

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

Architecture Rule B-2: “Memory accesses that are actually misaligned are not guaranteed to be atomic as observed from other threads, processors, and I/O devices.” on page 771.

The rules in this section provide further detail.

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

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

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

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

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

E.g. a misaligned access that is not split may be performed as a single access.

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

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

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

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

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

See section B.6 “Misalignment and MSA vector memory accesses” on page 781, for a discussion of element atomicity as applies to misaligned MSA vector memory accesses.

B.4.6.3 Misaligneds and Multiprocessor Memory Ordering

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

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

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

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

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

B.5 Pseudocode

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

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

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

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

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

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

```
endfor
```

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

B.5.1 Pseudocode distinguishing Actually Aligned from Actually Misaligned

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

Figure B.1 LoadPossiblyMisaligned / StorePossiblyMisaligned pseudocode

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

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

B.5.2 Actually Aligned

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

Figure B.2 LoadAligned / StoreAligned pseudocode

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

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

B.5.3 Byte Swapping

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

Figure B.3 LoadRawMemory Pseudocode Function

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

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

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

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

endfunction LoadRawMemory
```

Figure B.4 StoreRawMemory Pseudocode Function

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

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

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

endfunction StoreRawMemory
```


Helper functions for byte swapping according to endianness:

Figure B.5 Byteswapping pseudocode functions

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

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

B.5.4 Pseudocode Expressing Most General Misaligned Semantics

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

Figure B.6 LoadMisaligned most general pseudocode

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

B.5.5 Example Pseudocode for Possible Implementations

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

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

B.5.5.1 Example Byte-by-byte Pseudocode

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

This pseudocode tries to indicate that it is permissible to use such a 2-phase approach in an exception handler to emulate misaligneds in software. It is not acceptable to use a single pass of byte by byte stores, unless split stores half performed can be withdrawn, transactionally. But it is not required to save the translations of the first pass to reuse in the second pass (which would be extremely slow). If virtual addresses translations or

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

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

```
    return inbytes[]
endfunction LoadMisaligned

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

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

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

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

B.6 Misalignment and MSA vector memory accesses

B.6.1 Semantics

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

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

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

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

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

On MIPS32r5 16 and 32-bit scalar accesses are defined to be atomic, so e.g. each of the 32-bit elements of word vector loaded using LD.W would be atomic. However, on MIPS32r5 64 bit accesses are not defined to be atomic, so LD.D would not have element atomicity.

On MIPS64r5 16, 32, and 64 bit scalar accesses are atomic. So vector LD.H, LD.W, LD.D, and the corresponding stores would be element atomic.

5. MIPS® Architecture Reference Manual, Volume IV-j: The MIPS32® SIMD, Architecture Module. Document Number MD00867, Revision 1.05, June 21, 2013.

All of the rules in sections B.4.2 “Permissions and misaligned memory accesses”, B.4.4 “TLBs and Misaligned Memory Accesses”, B.4.5 “Memory Types and Misaligned Memory Accesses”, and B.4.6.1 “Misaligneds are Single Thread Atomic” apply to the vector load or store instructions as a whole.

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

E.g. misaligned vector memory references that partly fall outside the virtual address space are UNPREDICTABLE.

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

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

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

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

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

B.6.2 Pseudocode for MSA memory operations with misalignment

The MSA specification uses the following pseudocode functions to access memory:

Figure B.8 LoadTYPEVector / StoreTYPEVector used by MSA specification

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

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

where TYPE = Byte, Halfword, Word, Doubleword,
e.g. LoadByteVector, LoadHalfwordVector, etc.
```

These can be expressed in terms of the misaligned pseudocode operations as follows - passing the TYPE (Byte, Halfword, Word, DoubleWord) as a parameter:

Figure B.9 Pseudocode for LoadVector

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

Figure B.10 Pseudocode for StoreVector

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

Appendix C

Revision History

Revision	Date	Description
0.90	November 1, 2000	Internal review copy of reorganized and updated architecture documentation.
0.91	November 15, 2000	Internal 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	Update based on all review feedback: • 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 Release 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 updated, 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 cannot 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 simplifies 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. •
2.60	June 25, 2008	Changes in this revision: • Applied the new B0.01 template. • Update RDHWR description with the UserLocal register. • added PAUSE instruction • Ordering SYNCs • CMP behavior of CACHE, PREF*, SYNCI • DCLZ, DCLO operations was inverted • CVT.S.PL, CVT.S.PU are non-arithmetic (no exceptions) • *MADD.fmt & *MSUB.fmt are non-fused. • various typos fixed
2.61	July 10, 2008	• Revision History file was incorrectly copied from Volume III. • Removed index conditional text from PAUSE instruction description. • SYNC instruction - added additional format “SYNC stype”
2.62	January 2, 2009	• LWC1, LWXC1 - added statement that upper word in 64bit registers are UNDEFINED. • CVT.S.PL and CVT.S.PU descriptions were still incorrectly listing IEEE exceptions. • Typo in CFC1 Description. • CCRes is accessed through $3 for RDHWR, not $4.
3.00	March 25, 2010	• JALX instruction description added. • Sub-setting rules updated for JALX. • Implementation comment for DSP ASE compatibility.
3.01	June 01, 2010	• Copyright page updated. • User mode instructions not allowed to produce UNDEFINED results, only UNPREDICTABLE results.
3.02	March 21, 2011	• RECIP, RSQRT instructions do not require 64-bit FPU. • MADD/MSUB/NMADD/NMSUB pseudo-code was incorrect for PS format check.

Revision	Date	Description
3.50	September 20, 2012	• Added EVA load/store instructions: LBE, LBUE, LHE, LHUE, LWE, SBE, SHE, SWE, CACHEE, PREFE, LLE, SCE, LWLE, LWRE, SWLE, SWRE. • TLBWI - can be used to invalidate the VPN2 field of a TLB entry. • FCSR.MAC2008 bit affects intermediate rounding in MADD.fmt, MSUB.fmt, NMADD.fmt and NMSUB.fmt. • FCSR.ABS2008 bit defines whether ABS.fmt and NEG.fmt are arithmetic or not (how they deal with QNAN inputs).
3.51	October 20, 2012	• CACHE and SYNCI ignore RI and XI exceptions. • CVT, CEIL, FLOOR, ROUND, TRUNC to integer can't generate FP-Overflow exception.
5.00	December 14, 2012	• R5 changes: DSP and MT ASEs -> Modules • NMADD.fmt, NMSUB.fmt - for IEEE2008 negate portion is arithmetic.
5.01	December 15, 2012	• No technical content changes: • Update logos on Cover. • Update copyright page.
5.02	April 22, 2013	• Fix: Figure 2.26 Are64BitFPOperationsEnabled Pseudcode Function - "Enabled" was missing. • R5 change retroactive to R3: removed FCSR.MCA2008 bit: no architectural support for fused multiply add with no intermediate rounding. Applies to MADD.fmt, MSUB.fmt, NMADD.fmt, NMSUB.fmt. • Clarification: references to "16 FP registers mode" changed to "the FR=0 32-bit register model"; specifically, paired single (PS) instructions and long (L) format instructions have UNPREDICTABLE results if FR=0, as well as LUXC1and SUXC1. • Clarification: C.cond.fmt instruction page: cond bits 2..1 specify the comparison, cond bit 0 specifies ordered versus unordered, while cond bit 3 specifies signaling versus non-signaling. • R5 change: UFR (User mode FR change): CFC1, CTC1 changes.

Revision	Date	Description
5.03	August 21, 2013	• Resolved inconsistencies with regards to the availability of instructions in MIPS32r2: MADD.fmt family (MADD.S, MADD.D, NMADD.S, NMADD.D, MSUB.S, MSUB.D, NMSUB,S, NMSUB.D), RECIP.fmt family (RECIP.S, RECIP.D, RSQRT.S, RSQRT.D), and indexed FP loads and stores (LWXC1, LDXC1, SWXC1, SDXC1). The appendix section A.2 “Instruction Bit Encoding Tables”, shared between Volume I and Volume II of the ARM, was updated, in particular the new upright delta Δ mark is added to Table A.2 “Symbols Used in the Instruction Encoding Tables”, replacing the inverse delta marking ∇ for these instructions. Similar updates made to microMIPS’s corresponding sections. Instruction set descriptions and pseudocode in Volume II, Basic Instruction Set Architecture, updated. These instructions are required in MIPS32r2 if an FPU is implemented. . • Misaligned memory access support for MSA: see Volume II, Appendix B “Misaligned Memory Accesses”. • Has2008 is required as of release 5 - Table 5.4, “FIR Register Descriptions”. • ABS2008 and NAN2008 fields of Table 5.7 “FCSR RegisterField Descriptions” were optional in release 3 and could be R/W, but as of release 5 are required, read-only, and preset by hardware. • FPU FCSR.FS Flush Subnormals / Flush to Zero behavior is made consistent with MSA behavior, in MSACSR.FS: Table 5.7, “FCSR Register Field Descriptions”, updated. New section 5.8.1.4 “Alternate Flush to Zero Underflow Handling”. • Volume I, Section 2.2 “Compliance ad Subsetting” noted that the L format is required in MIPS FPUs, to be consistent with Table 5.4 “FIR Register Field Definitions” . • Noted that UFR and UNFR can only be written with the value 0 from GPR[0]. See section 5.6.5 “User accessible FPU Register model control (UFR, CP1 Control Register 1)” and section 5.6.5 “User accessible Negated FPU Register model control (UNFR, CP1 Control Register 4)”
5.04	December 11, 2013	LLSC Related Changes • Added ERETNC. New. • Modified SC handling: refined, added, and elaborated cases where SC can fail or was UNPREDICTABLE. XPA Related Changes • Added MTHC0, MFHC0 to access extensions. All new. • Modified MTC0 for MIPS32 to zero out the extended bits which are writable. This is to support compatibility of XPA hardware with non XPA software. In pseudo-code, added registers that are impacted. • MTHC0 and MFHC0 - Added RI conditions.

Revision	Date	Description
6.00 - R6U draft	Dec. 19, 2013	• Feature complete R6U draft of Volume II new instructions.
	Jan 14-16, 2014	• Split MAX.fmt-family, instruction description that described multiple instructions, into separate instruction description pages MAX.fmt, MAX_A.fmt, MIN.fmt, MIN_A.fmt. • Mnemonic change: AUIPA changed to ALUIPC, Aligned Add Upper Immediate to PC. Now all MIPS32 Release 6 new PC relative instructions end in "P". • Renamed CMP.cond.fmt -> CMP.condn.fmt, i.e. renamed 5-bit `cond` field "`condn`" to distinguish it from old 4-bit `cond` field. • Cleaning up descriptions of NAL and BAL to reduce confusion about deprecation versus removal of BLTZAL and BGEZAL. • DAHI and DATI use `rs src/dest` register, not `rt`. • Table showing that the compact branches are complete, reversing rs and rt for BLEC, BGTC, BLEUC, BGTUC • Forbidden slot RI required; takes exception like delay slot; boilerplate consistency automated. • MOD instruction family: remainder has same sign as dividend • Updated to R6U 1.03
	Jan 17, 2014	• NAL, BAL: improved confusing explanation of how NAL and BAL used to be special cases of BLEZAL, etc., instructions removed by MIPS32 Release 6 • Forbidden Slot boilerplate: requires Reserved Instruction exception for control instructions, even if interrupted: exception state (EPC, etc.) points to branch, not Forbidden Slot, like delay slot.
	Jan 20, 2014	• Fixed bugs and changed instruction encodings: BEQZALC, BNEZALC, BGEUC, BLTUC, BLEZLC family, BC1EQZ, BC2EQZ, BC1NEZ, BC2NEZ, BITSWAP • AUI, BAL
R6U draft	Feb 10, 2014	• Refactored "Compatibility and Subsetting" sections of Volumes I and II for reuse without replication. • Updated Volume II tables of instructions by categories (preceding section entitled Alphabetical List of Instructions) for R6U changes.
R6U-pre-release draft	Feb. 11, 2014	Technical Publications preparing for release. **Summary of all R6U drafts up to this date - R6U version 1.03** • MIPS3D removed from the MIPS32 Release 6 architecture. • Some 3-source instructions (conditional moves) replaced with new 2-source instructions: MOVZ/MOVN.fmt replaced by SELEQZ/SELNEZ.fmt; MOVZ/MOVN replaced by SELEQZ/SELNEZ. • PREF/PREFE: Unsound prefetch hints downgraded; optional implementation dependent prefetch hints expanded. Free up Opcode Space • Change encodings of LL/SC/LLD/SCD/PREF/CACHE, reducing offset from 16 bits to 9 bits • SPECIAL2 encodings changed: CLO/CLZ/DCLO/DCLZ • Other changes mentioned below: traps with immediate operands removed (ADDI/DADDI, TGEI/TGEIU/TLTI/TLTIU/TEQI/TNEI) • Free 15 major opcodes: COP1X, SPECIAL2, LWL/LWR, SWL/SWR, LDL/LDR, SDL/SDR, LL/SC, LLD/SCD, PREF, CACHE, as described below, by changing encodings.

Revision	Date	Description
		Integer Multiply and Divide • Integer accumulators (HI/LO) removed from base MIPS32 Release 6, moved to DSPr6, allowed only with microMIPS: MFHI, MTHIO, MFLO, MTLO, MADD, MADDU, MUL, MSUB, MSUBU removed. • MIPS32 Release 6 adds multiply and divide instructions that write to same-width register: MULT replaced by MUL/MUH; MULTU replaced by MULU/MUHU; DIV replaced by DIV/MOD; DIVU replaced by DIVU/MODU; similarly for 64-bit DMUH, etc. Control Transfer Instructions (CTIs) • Branch likely instructions removed by MIPS32 Release 6: BEQL, etc. • Enhanced compact branches and jumps provided • No delay slots; back-to-back branches disallowed (forbidden slot) • More complete set of conditions: BEQC/BNEC, all signed and unsigned reg-reg comparisons, e.g. BLTC, BLTUC; all comparisons against zero, e.g. BLTZC • More complete set of conditional procedure call instructions: BEQZALC, BNEZALC • Large offset PC-relative branches: BC/BALC 26-bit offset (scaled by 4); BEQZC/BNEZC 21-bit offset • JIC/JIALC: “indexed” jumps, jump to register + sign extended 16-bit offset • Trap-in-overflow adds with immediate removed by MIOPSr6: ADDI, DADDI; replaced by branches on overflow BOVC/BNVC. • Redundant JR.HB removed, aliased to JALR.HB with rdest=0. • BLTZAL/BGEZAL removed; not used because unconditionally wrote link register SSNOP identical to NOP. Misaligned Memory Accesses • Unaligned load/store instructions (LWL/LWR, etc.) removed from MIPS32 Release 6. Support for misaligned memory accesses must be provided by a MIPS32 Release 6 system for all ordinary loads and stores, by hardware or by software trap-and-emulate. • CPU scalar ALIGN instruction Address Generation and Constant Building • Instructions to build large constants (such as address constants): AUI (Add upper immediate), DAHI, DATI. • Instructions for PC-relative address formation: ADDIUPC, AUIPC, ALUIPC. • PC-relative loads: LWP, LWUP, LDP. • Indexed FPU memory accesses removed: LWXC1, LUXC1, PFX, etc. • Load-scaled-address instructions: LSA, DLSA • 32-bit address wrapping improved. DSP Module • DSP Module and SmartMIPS disallowed; recommend MSA instead • DSPr6 to be defined, used with microMIPS. • Instructions promoted from DSP Module to Base ISA: BALIGN becomes MIPS32 Release 6 ALIGN, BITREV becomes MIPS32 Release 6 BITSWAP

Revision	Date	Description
		FPU and co-processor • Instruction encodings changed: COP2 loads/stores, cache/prefetch, SPECIAL2: LWC2/SWC2, LDC2/SWC2 • FR=0 not allowed, FR=1 required. • Compatibility and Subsetting section amended to allow a single precision only FPU (FIR.S=FIR.W=1, FIR.D=FIR.L=0.) • Paired Single (PS) removed from the MIPS32 Release 6 architecture, including: COP1.PS, COP1X.PS, BC1ANY2, BC1ANY4, CVT.PS.S, CVT.PS.W. • FPU scalar counterparts to MSA instructions: RINT.fmt, CLASS.fmt, MAX/MAXA/MIN/MINA.fmt. • Unfused multiply adds removed: MADD/MSUB/NMADD/NMSUB.fmt • IEEE2008 Fused multiply adds added: MADDF/MSUBF.fmt • Floating point condition codes and related instructions removed: C.cond.fmt removed, BC1T/BC1F, MOVF/MOVT. • MOVF/MOVT.fmt replaced by SEL.fmt • New FP compare instruction CMP.cond.fmt places result in FPR and related BC1EQZ/BC2EQZ • New FP comparisons: CMP.cond.fmt with `cond` = OR (ordered), UNE (Unordered or Not Equal), NE (Not Equal). • Coprocessor 2 condition codes removed: BC2F/BC2T removed, replaced by BC2NEQZ/BC2EQZ **Recent R6U architecture changes not fully reflected in this draft:** • This draft does not completely reflect the new 32-bit address wrapping proposal but still refers in some places to the old IAM (Implicit Address Mode) proposal. • This draft does not yet reflect constraints on endianness, in particular in the section ion Misaligned memory access support: e.g. code and data must have the same endianness, Status.RE is removed, etc. • BC1EQZ/BC1NEZ will test only bit 0 of the condition register, not all bits. • This draft does not yet say that writing to a 32-bit FPR renders upper bits of a 64 bit FPR or 128 bit floating point register UNPREDICTABLE; it describes the old proposal of zeroing the upper bits.

Revision	Date	Description
		Known issues: • NAL should not be in the list of instructions which cause RI when placed in the delay or forbidden slot. • MIN/MAX/MINA/MAXA.fmt are R6 instructions that will be included in next rev of spec. • This draft describes MIPS32 Release 6, as well as earlier releases of the MIPS architecture. E.g. instructions that were present in MIPS32 Release 5 but which were removed in MIPS32 Release 6 are still in the manual, although they should be clearly marked “as removed” in the Availability section. • R6U new instruction pseudocode is 64-bit, rather than 32-bit, albeit attempting to use notations that apply to both. • Certain new instruction descriptions are “unsplit”, describing families of instructions such as all compact branches, rather than separate descriptions of each instruction. This facilitates comparison and consistency, but currently allows certain instructions to appear inappropriately in the MIPS32 Release 6 manual. A future release of the manual will “split” these instruction family descriptions, e.g. the compact branch family will be split up into at least 12 different instruction descriptions. • R6U requires misalignment support for all ordinary memory reference instructions, but the pseudocode does not yet reflect this. Boilerplate has been added to all existing instructions saying this. • The new R6U PC-relative loads (LWP, LWUP, LDP) in this draft incorrectly say that misaligned accesses are permitted.
R6U-pre-release draft	Feb. 13, 2014	• ALIGN/DALIGN: clarified bp=0 behavior • ALIGN/DALIGN pseudocode used || as logical OR rather than MIPS’ pseudocode concatenate. • Removed incorrect note about not using r31 as a source register to BAL. • MIPS32 Release 6 requires BC1EQZ/BC1NEZ if an FPU is present, i.e. they cannot signal RI. • R6U 1.05 change: BC1EQZ/BC1NEZ test only bit 0 of the FPY; changed from testing if any bit nonzero; helps with trap-and-emulate of DP on an SP-only FPU. • Known problem: R6U 1.05 change not yet made: all 32-bit FP operations leave upper bits of 64 bit FOR and./or 128-bit MSR unpredictable; helps with trap-and-emulate of DP on an SP-only FPU. • Clearly marked all .PS instructions as removed via -MIPSr6 in instruction format. • DMUL, DMULTU, DDIV, DDIVU marked “as removed in Release 6”. • Started using =MIPSr6 notation to indicate that an instruction has been changed but is still present. JR.HB =MIPSr6, aliased to JALR.HB. SSNOP =MIPSr6, treated as NOP. • Noted that BLTZAL and BGEZAL are removed by MIPS32 Release 6, the special cases NAL=BLTZAL with rs=0 and BAL=BGEZAL with rs=0, remain supported by MIPS32 Release 6. • Marked conditional traps with immediate “as removed in Release 6”. • Overeager propagation of r31 restriction to non-call instructions5 removed. • Emphasized that unconditional compact CTIs have neither Delay Slot nor Forbidden Slot. • SDBBP updated for R6P facility to disable if no hardware debug trap handler • UFR/UNFR (User-mode FR facility) disallowed in MIPS32 Release 6: changes to CTC1 and CFC1 instructions.

Revision	Date	Description
R6U ARM Volume II 6.00 preliminary release	February 14, 2014	• Last minute change: BC1EQZ.fmt and BC1NEZ.fmt test only bit 0, least significant bit, of FPR. Known issues: • Similar changes to SEL.fmt, SELEQZ.fmt, SELNEZ.fmt not yet made.
post-6.00	February 20, 2014	• FPU truth consuming instructions (BC1EQZ.fmt, BC1NEZ.fmt, SEL.fmt, SELEQZ.fmt, SELNEZ.fmt) change completed: test bit 0, least-significant-bit, of FPR containing condition.
post-6.00	April 1, 2014	• Standard boilerplate listing instructions not allowed in branch delay slots or forbidden slots added to all individual instruction descriptions. Not a technical change; just ensuring consistency.