



# **MIPS32™ Architecture For Programmers**

## **Volume II: The MIPS32™ Instruction Set**

**Document Number: MD00086**

**Revision 1.00**

**August 29, 2002**

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

**Copyright © 2001-2002 MIPS Technologies Inc. All rights reserved.**

Copyright © 2001-2002 MIPS Technologies, Inc. All rights reserved.

Unpublished rights (if any) are reserved under the Copyright Laws of the United States of America.

If this document is provided in source format (i.e., in a modifiable form such as in FrameMaker or Microsoft Word format), then its use and distribution is subject to a written agreement with MIPS Technologies, Inc. ("MIPS Technologies"). UNDER NO CIRCUMSTANCES MAY A DOCUMENT PROVIDED IN SOURCE FORMAT BE DISTRIBUTED TO A THIRD PARTY WITHOUT THE EXPRESS WRITTEN CONSENT OF MIPS TECHNOLOGIES.

This document contains information that is proprietary to MIPS Technologies. Any copying, reproducing, modifying, or use of this information (in whole or in part) which is not expressly permitted in writing by MIPS Technologies or a contractually-authorized third party is strictly prohibited. At a minimum, this information is protected under unfair competition and copyright laws. Violations thereof may result in criminal penalties and fines.

MIPS Technologies or any contractually-authorized third party reserves the right to change the information contained in this document to improve function, design or otherwise. MIPS Technologies does not assume any liability arising out of the application or use of this information, or of any error of omission in such information. Any warranties, whether express, statutory, implied or otherwise, including but not limited to the implied warranties of merchantability or fitness for a particular purpose, are excluded. Any license under patent rights or any other intellectual property rights owned by MIPS Technologies or third parties shall be conveyed by MIPS Technologies or any contractually-authorized third party in a separate license agreement between the parties.

The information contained in this document shall not be exported or transferred for the purpose of reexporting in violation of any U.S. or non-U.S. regulation, treaty, Executive Order, law, statute, amendment or supplement thereto.

The information contained in this document constitutes one or more of the following: commercial computer software, commercial computer software documentation or other commercial items. If the user of this information, or any related documentation of any kind, including related technical data or manuals, is an agency, department, or other entity of the United States government ("Government"), the use, duplication, reproduction, release, modification, disclosure, or transfer of this information, or any related documentation of any kind, is restricted in accordance with Federal Acquisition Regulation 12.212 for civilian agencies and Defense Federal Acquisition Regulation Supplement 227.7202 for military agencies. The use of this information by the Government is further restricted in accordance with the terms of the license agreement(s) and/or applicable contract terms and conditions covering this information from MIPS Technologies or any contractually-authorized third party.

MIPS<sup>®</sup>, R3000<sup>®</sup>, R4000<sup>®</sup>, R5000<sup>®</sup> and R10000<sup>®</sup> are among the registered trademarks of MIPS Technologies, Inc. in the United States and certain other countries, and MIPS16<sup>™</sup>, MIPS16e<sup>™</sup>, MIPS32<sup>™</sup>, MIPS64<sup>™</sup>, MIPS-3D<sup>™</sup>, MIPS-based<sup>™</sup>, MIPS I<sup>™</sup>, MIPS II<sup>™</sup>, MIPS III<sup>™</sup>, MIPS IV<sup>™</sup>, MIPS V<sup>™</sup>, MDMX<sup>™</sup>, MIPSsim<sup>™</sup>, MIPSsimCA<sup>™</sup>, MIPSsimIA<sup>™</sup>, QuickMIPS<sup>™</sup>, SmartMIPS<sup>™</sup>, MIPS Technologies logo, 4K<sup>™</sup>, 4Kc<sup>™</sup>, 4Km<sup>™</sup>, 4Kp<sup>™</sup>, 4KE<sup>™</sup>, 4KEc<sup>™</sup>, 4KEm<sup>™</sup>, 4KEp<sup>™</sup>, 4KS<sup>™</sup>, 4KSc<sup>™</sup>, M4K<sup>™</sup>, 5K<sup>™</sup>, 5Kc<sup>™</sup>, 5Kf<sup>™</sup>, 20K<sup>™</sup>, 20Kc<sup>™</sup>, 25Kf<sup>™</sup>, R4300<sup>™</sup>, ASMACRO<sup>™</sup>, ATLAS<sup>™</sup>, BusBridge<sup>™</sup>, CoreFPGA<sup>™</sup>, CoreLV<sup>™</sup>, EC<sup>™</sup>, JALGO<sup>™</sup>, MALTA<sup>™</sup>, MGB<sup>™</sup>, PDtrace<sup>™</sup>, SEAD<sup>™</sup>, SEAD-2<sup>™</sup>, SOC-it<sup>™</sup>, The Pipeline<sup>™</sup>, and YAMON<sup>™</sup> are among the trademarks of MIPS Technologies, Inc.

All other trademarks referred to herein are the property of their respective owners.

# Table of Contents

Chapter 1 About This Book .....	1
1.1 Typographical Conventions .....	1
1.1.1 Italic Text .....	1
1.1.2 Bold Text .....	1
1.1.3 Courier Text .....	1
1.2 UNPREDICTABLE and UNDEFINED .....	2
1.2.1 UNPREDICTABLE .....	2
1.2.2 UNDEFINED .....	2
1.3 Special Symbols in Pseudocode Notation .....	2
1.4 For More Information .....	4
Chapter 2 Guide to the Instruction Set .....	7
2.1 Understanding the Instruction Fields .....	7
2.1.1 Instruction Fields .....	8
2.1.2 Instruction Descriptive Name and Mnemonic .....	9
2.1.3 Format Field .....	9
2.1.4 Purpose Field .....	10
2.1.5 Description Field .....	10
2.1.6 Restrictions Field .....	10
2.1.7 Operation Field .....	11
2.1.8 Exceptions Field .....	11
2.1.9 Programming Notes and Implementation Notes Fields .....	11
2.2 Operation Section Notation and Functions .....	12
2.2.1 Instruction Execution Ordering .....	12
2.2.2 Pseudocode Functions .....	12
2.3 Op and Function Subfield Notation .....	20
2.4 FPU Instructions .....	20
Chapter 3 The MIPS32™ Instruction Set .....	21
3.1 Compliance and Subsetting .....	21
3.2 Alphabetical List of Instructions .....	22
ABS.fmt .....	30
ADD .....	31
ADD.fmt .....	32
ADDI .....	33
ADDIU .....	34
ADDU .....	35
AND .....	36
ANDI .....	37
B .....	38
BAL .....	39
BC1F .....	40
BC1FL .....	42
BC1T .....	44
BC1TL .....	46
BC2F .....	48
BC2FL .....	49
BC2T .....	51
BC2TL .....	52
BEQ .....	54
BEQL .....	55
BGEZ .....	57
BGEZAL .....	58
BGEZALL .....	59
BGEZL .....	61

BGTZ.....	63
BGTZL.....	64
BLEZ.....	66
BLEZL.....	67
BLTZ.....	69
BLTZAL.....	70
BLTZALL.....	71
BLTZL.....	73
BNE.....	75
BNEL.....	76
BREAK.....	78
C.cond.fmt.....	79
CACHE.....	84
CEIL.W.fmt.....	91
CFC1.....	92
CFC2.....	94
CLO.....	95
CLZ.....	96
COP2.....	97
CTC1.....	98
CTC2.....	100
CVT.D.fmt.....	101
CVT.S.fmt.....	102
CVT.W.fmt.....	103
DERET.....	104
DIV.....	106
DIV.fmt.....	108
DIVU.....	109
ERET.....	110
FLOOR.W.fmt.....	111
J.....	112
JAL.....	113
JALR.....	114
JR.....	116
LB.....	118
LBU.....	119
LDC1.....	120
LDC2.....	121
LH.....	122
LHU.....	123
LL.....	124
LUI.....	126
LW.....	127
LWC1.....	128
LWC2.....	129
LWL.....	130
LWR.....	133
MADD.....	137
MADDU.....	138
MFC0.....	139
MFC1.....	140
MFC2.....	141
MFHI.....	142
MFLO.....	143
MOV.fmt.....	144
MOVF.....	145
MOVF.fmt.....	146
MOVN.....	148
MOVN.fmt.....	149
MOVT.....	151
MOVT.fmt.....	152
MOVZ.....	154
MOVZ.fmt.....	155
MSUB.....	157
MSUBU.....	158
MTC0.....	159

MTC1 .....	160
MTC2 .....	161
MTHI .....	162
MTLO .....	163
MUL .....	164
MUL.fmt .....	165
MULT .....	166
MULTU .....	167
NEG.fmt .....	168
NOP .....	169
NOR .....	170
OR .....	171
ORI .....	172
PREF .....	173
ROUND.W.fmt .....	177
SB .....	179
SC .....	180
SDBBP .....	183
SDC1 .....	184
SDC2 .....	185
SH .....	186
SLL .....	187
SLLV .....	188
SLT .....	189
SLTI .....	190
SLTIU .....	191
SLTU .....	192
SQRT.fmt .....	193
SRA .....	194
SRAV .....	195
SRL .....	196
SRLV .....	197
SSNOP .....	198
SUB .....	199
SUB.fmt .....	200
SUBU .....	201
SW .....	202
SWC1 .....	203
SWC2 .....	204
SWL .....	205
SWR .....	207
SYNC .....	209
SYSCALL .....	213
TEQ .....	214
TEQI .....	215
TGE .....	216
TGEI .....	217
TGEIU .....	218
TGEU .....	219
TLBP .....	220
TLBR .....	221
TLBWI .....	223
TLBWR .....	225
TLT .....	227
TLTI .....	228
TLTIU .....	229
TLTU .....	230
TNE .....	231
TNEI .....	232
TRUNC.W.fmt .....	233
WAIT .....	235
XOR .....	237
XORI .....	238
Appendix A Instruction Bit Encodings .....	239
A.1 Instruction Encodings and Instruction Classes .....	239

---

A.2 Instruction Bit Encoding Tables.....	239
A.3 Floating Point Unit Instruction Format Encodings .....	244
Appendix B Revision History .....	245

---

## List of Figures

Figure 2-1: Example of Instruction Description .....	8
Figure 2-2: Example of Instruction Fields .....	9
Figure 2-3: Example of Instruction Descriptive Name and Mnemonic .....	9
Figure 2-4: Example of Instruction Format.....	9
Figure 2-5: Example of Instruction Purpose .....	10
Figure 2-6: Example of Instruction Description .....	10
Figure 2-7: Example of Instruction Restrictions .....	11
Figure 2-8: Example of Instruction Operation .....	11
Figure 2-9: Example of Instruction Exception .....	11
Figure 2-10: Example of Instruction Programming Notes .....	12
Figure 2-11: COP_LW Pseudocode Function.....	13
Figure 2-12: COP_LD Pseudocode Function.....	13
Figure 2-13: COP_SW Pseudocode Function .....	13
Figure 2-14: COP_SD Pseudocode Function.....	14
Figure 2-15: AddressTranslation Pseudocode Function .....	14
Figure 2-16: LoadMemory Pseudocode Function.....	15
Figure 2-17: StoreMemory Pseudocode Function .....	15
Figure 2-18: Prefetch Pseudocode Function .....	16
Figure 2-19: ValueFPR Pseudocode Function .....	17
Figure 2-20: StoreFPR Pseudocode Function .....	17
Figure 2-21: SyncOperation Pseudocode Function.....	18
Figure 2-22: SignalException Pseudocode Function .....	18
Figure 2-23: SignalDebugBreakpointException Pseudocode Function.....	18
Figure 2-24: SignalDebugModeBreakpointException Pseudocode Function .....	18
Figure 2-25: NullifyCurrentInstruction PseudoCode Function.....	19
Figure 2-26: CoprocessorOperation Pseudocode Function.....	19
Figure 2-27: JumpDelaySlot Pseudocode Function .....	19
Figure 2-28: FPConditionCode Pseudocode Function.....	20
Figure 2-29: SetFPConditionCode Pseudocode Function.....	20
Figure 3-1: Usage of Address Fields to Select Index and Way .....	85
Figure 3-2: Unaligned Word Load Using LWL and LWR .....	130
Figure 3-3: Bytes Loaded by LWL Instruction .....	131
Figure 3-4: Unaligned Word Load Using LWL and LWR .....	134
Figure 3-5: Bytes Loaded by LWL Instruction .....	135
Figure 3-6: Unaligned Word Store Using SWL and SWR .....	205
Figure 3-7: Bytes Stored by an SWL Instruction .....	206
Figure 3-8: Unaligned Word Store Using SWR and SWL .....	207
Figure 3-9: Bytes Stored by SWR Instruction .....	208
Figure A-1: Sample Bit Encoding Table.....	240

---

## List of Tables

Table 1-1: Symbols Used in Instruction Operation Statements .....	2
Table 2-1: AccessLength Specifications for Loads/Stores .....	16
Table 3-1: CPU Arithmetic Instructions .....	22
Table 3-2: CPU Branch and Jump Instructions.....	22
Table 3-3: CPU Instruction Control Instructions .....	23
Table 3-4: CPU Load, Store, and Memory Control Instructions .....	23
Table 3-5: CPU Logical Instructions .....	24
Table 3-6: CPU Move Instructions .....	24
Table 3-7: CPU Shift Instructions .....	24
Table 3-8: CPU Trap Instructions .....	25
Table 3-9: Obsolete CPU Branch Instructions .....	25
Table 3-10: FPU Arithmetic Instructions.....	26
Table 3-11: FPU Branch Instructions.....	26
Table 3-12: FPU Compare Instructions .....	26
Table 3-13: FPU Convert Instructions .....	26
Table 3-14: FPU Load, Store, and Memory Control Instructions.....	27
Table 3-15: FPU Move Instructions.....	27
Table 3-16: Obsolete FPU Branch Instructions .....	27
Table 3-17: Coprocessor Branch Instructions.....	27
Table 3-18: Coprocessor Execute Instructions.....	27
Table 3-19: Coprocessor Load and Store Instructions .....	28
Table 3-20: Coprocessor Move Instructions .....	28
Table 3-21: Obsolete Coprocessor Branch Instructions.....	28
Table 3-22: Privileged Instructions .....	28
Table 3-23: EJTAG Instructions .....	29
Table 3-24: FPU Comparisons Without Special Operand Exceptions .....	80
Table 3-25: FPU Comparisons With Special Operand Exceptions for QNaNs .....	81
Table 3-26: Usage of Effective Address .....	84
Table 3-27: Encoding of Bits[17:16] of CACHE Instruction .....	85
Table 3-28: Encoding of Bits [20:18] of the CACHE Instruction .....	86
Table 3-29: Values of the <i>hint</i> Field for the PREF Instruction .....	174
Table A-1: Symbols Used in the Instruction Encoding Tables .....	240
Table A-2: MIPS32 Encoding of the Opcode Field .....	241
Table A-3: MIPS32 SPECIAL Opcode Encoding of Function Field .....	241
Table A-4: MIPS32 REGIMM Encoding of rt Field .....	241
Table A-5: MIPS32 SPECIAL2 Encoding of Function Field.....	242
Table A-6: MIPS32 MOVCI Encoding of tf Bit.....	242
Table A-7: MIPS32 COPz Encoding of rs Field.....	242
Table A-8: MIPS32 COPz Encoding of rt Field When rs=BCz .....	242
Table A-9: MIPS32 COP0 Encoding of rs Field.....	242
Table A-10: MIPS32 COP0 Encoding of Function Field When rs=CO .....	243
Table A-11: MIPS32 COP1 Encoding of rs Field.....	243
Table A-12: MIPS32 COP1 Encoding of Function Field When rs=S .....	243
Table A-13: MIPS32 COP1 Encoding of Function Field When rs=D.....	243
Table A-14: MIPS32 COP1 Encoding of Function Field When rs=W .....	244
Table A-15: MIPS32 COP1 Encoding of tf Bit When rs=S, D, or PS, Function=MOVCF.....	244
Table 3-30: Floating Point Unit Instruction Format Encodings.....	244



---

## About This Book

The MIPS32™ Architecture For Programmers Volume II comes as a multi-volume set.

- Volume I describes conventions used throughout the document set, and provides an introduction to the MIPS32™ Architecture
- Volume II provides detailed descriptions of each instruction in the MIPS32™ instruction set
- Volume III describes the MIPS32™ Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS32™ processor implementation
- Volume IV-a describes the MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS32™ Architecture and is not applicable to the MIPS32™ document set
- Volume IV-c describes the MIPS-3D™ Application-Specific Extension to the MIPS64™ Architecture and is not applicable to the MIPS32™ document set
- Volume IV-d describes the SmartMIPS™ Application-Specific Extension to the MIPS32™ 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*, *registers*, that are important from a software perspective (for instance, address bits used by software, and programmable fields and registers), and various *floating point instruction formats*, such as *S*, *D*, and *PS*
- is used for the memory access types, such as *cached* and *uncached*

#### 1.1.2 Bold Text

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

#### 1.1.3 Courier Text

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

## 1.2 UNPREDICTABLE and UNDEFINED

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

### 1.2.1 UNPREDICTABLE

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

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

- Implementations of operations generating **UNPREDICTABLE** results must not depend on any data source (memory or internal state) which is inaccessible in the current processor mode
- **UNPREDICTABLE** operations must not read, write, or modify the contents of memory or internal state which is inaccessible in the current processor mode. For example, **UNPREDICTABLE** operations executed in user mode must not access memory or internal state that is only accessible in Kernel Mode or Debug Mode or in another process
- **UNPREDICTABLE** operations must not halt or hang the processor

### 1.2.2 UNDEFINED

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

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

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

## 1.3 Special Symbols in Pseudocode Notation

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

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

Symbol	Meaning
$\leftarrow$	Assignment
$=, \neq$	Tests for equality and inequality
$\parallel$	Bit string concatenation
$x^y$	A $y$ -bit string formed by $y$ copies of the single-bit value $x$

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

Symbol	Meaning
$b\#n$	A constant value $n$ in base $b$ . For instance $10\#100$ represents the decimal value 100, $2\#100$ represents the binary value 100 (decimal 4), and $16\#100$ represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10.
$x_{y..z}$	Selection of bits $y$ through $z$ of bit string $x$ . Little-endian bit notation (rightmost bit is 0) is used. If $y$ is less than $z$ , this expression is an empty (zero length) bit string.
$+$ , $-$	2's complement or floating point arithmetic: addition, subtraction
$*$ , $\times$	2's complement or floating point multiplication (both used for either)
$\text{div}$	2's complement integer division
$\text{mod}$	2's complement modulo
$/$	Floating point division
$<$	2's complement less-than comparison
$>$	2's complement greater-than comparison
$\leq$	2's complement less-than or equal comparison
$\geq$	2's complement greater-than or equal comparison
$\text{nor}$	Bitwise logical NOR
$\text{xor}$	Bitwise logical XOR
$\text{and}$	Bitwise logical AND
$\text{or}$	Bitwise logical OR
$\text{GPRLEN}$	The length in bits (32 or 64) of the CPU general-purpose registers
$\text{GPR}[x]$	CPU general-purpose register $x$ . The content of $\text{GPR}[0]$ is always zero.
$\text{FPR}[x]$	Floating Point operand register $x$
$\text{FCC}[CC]$	Floating Point condition code $CC$ . $\text{FCC}[0]$ has the same value as $\text{COC}[1]$ .
$\text{FPR}[x]$	Floating Point (Coprocessor unit 1), general register $x$
$\text{CPR}[z,x,s]$	Coprocessor unit $z$ , general register $x$ , select $s$
$\text{CP2CPR}[x]$	Coprocessor unit 2, general register $x$
$\text{CCR}[z,x]$	Coprocessor unit $z$ , control register $x$
$\text{CP2CCR}[x]$	Coprocessor unit 2, control register $x$
$\text{COC}[z]$	Coprocessor unit $z$ condition signal
$\text{Xlat}[x]$	Translation of the MIPS16e GPR number $x$ into the corresponding 32-bit GPR number
$\text{BigEndianMem}$	Endian mode as configured at chip reset ( $0 \rightarrow \text{Little-Endian}$ , $1 \rightarrow \text{Big-Endian}$ ). Specifies the endianness of the memory interface (see <code>LoadMemory</code> and <code>StoreMemory</code> pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution.
$\text{BigEndianCPU}$	The endianness for load and store instructions ( $0 \rightarrow \text{Little-Endian}$ , $1 \rightarrow \text{Big-Endian}$ ). In User mode, this endianness may be switched by setting the <i>RE</i> bit in the <i>Status</i> register. Thus, $\text{BigEndianCPU}$ may be computed as $(\text{BigEndianMem} \text{ XOR } \text{ReverseEndian})$ .

Table 1-1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
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 <i>RE</i> bit of the <i>Status</i> register. Thus, ReverseEndian may be computed as (SR <sub>RE</sub> and User mode).
<i>LLbit</i>	Bit of <b>virtual</b> state used to specify operation for instructions that provide atomic read-modify-write. <i>LLbit</i> is set when a linked load occurs; it is tested and cleared by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.
<b>I</b> , <b>I+n</b> , <b>I-n</b> :	<p>This occurs as a prefix to <i>Operation</i> 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 <b>I</b>. 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 <b>I</b>, 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 <b>I+1</b>.</p> <p>The effect of pseudocode statements for the current instruction labelled <b>I+1</b> appears to occur “at the same time” as the effect of pseudocode statements labeled <b>I</b> 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.</p>
PC	The <i>Program Counter</i> 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 <i>PC</i> during an instruction time. If no value is assigned to <i>PC</i> 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 <i>PC</i> during the instruction time of the instruction in the branch delay slot.
PABITS	The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{\text{PABITS}} = 2^{36}$ bytes.
FP32RegistersMode	<p>Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR.</p> <p>In MIPS32 implementations, <b>FP32RegistersMode</b> 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 <b>FP32RegistersMode</b> is computed from the FR bit in the <i>Status</i> 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.</p> <p>The value of <b>FP32RegistersMode</b> is computed from the FR bit in the <i>Status</i> register.</p>
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 <i>dynamic</i> state of the instruction, not the <i>static</i> state. That is, the value is false if a branch or jump occurs to an instruction whose PC immediately follows a branch or jump, but which is not executed in the delay slot of a branch or jump.
SignalException(exception, argument)	Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument). Control does not return from this pseudocode function - the exception is signaled at the point of the call.

## 1.4 For More Information

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

<http://www.mips.com>

Comments or questions on the MIPS32™ Architecture or this document should be directed to

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

or via E-mail to [architecture@mips.com](mailto:architecture@mips.com).



---

## Guide to the Instruction Set

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

### 2.1 Understanding the Instruction Fields

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

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

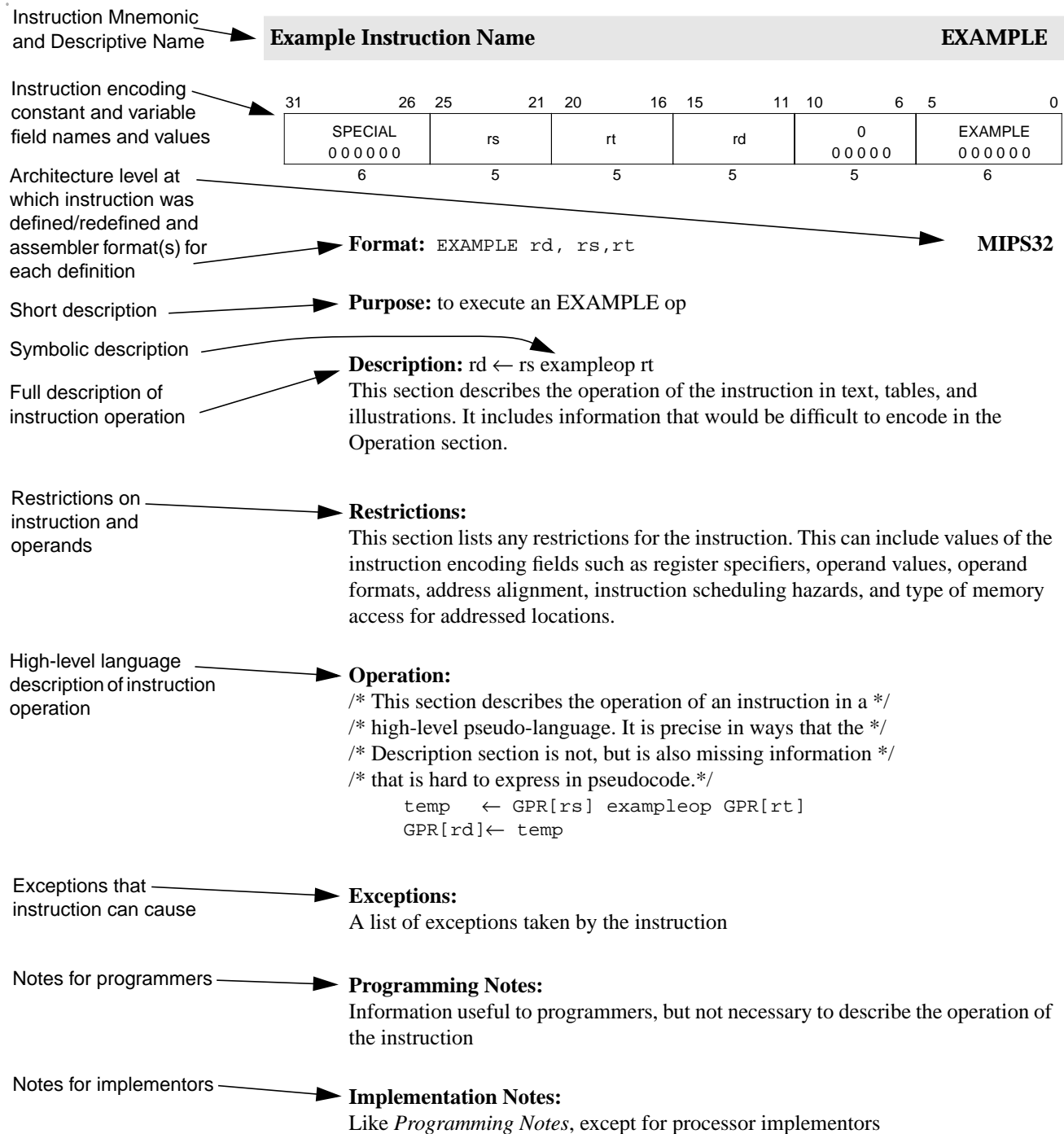


Figure 2-1 Example of Instruction Description

### 2.1.1 Instruction Fields

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



- The values of constant fields and the *opcode* names are listed in uppercase (SPECIAL and ADD in Figure 2-2). Constant values in a field are shown in binary below the symbolic or hexadecimal value.
- All variable fields are listed with the lowercase names used in the instruction description (*rs*, *rt* and *rd* in Figure 2-2).
- Fields that contain zeros but are not named are unused fields that are required to be zero (bits 10:6 in Figure 2-2). If such fields are set to non-zero values, the operation of the processor is **UNPREDICTABLE**.

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

Figure 2-2 Example of Instruction Fields

### 2.1.2 Instruction Descriptive Name and Mnemonic

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

Add Word	ADD
----------	-----

Figure 2-3 Example of Instruction Descriptive Name and Mnemonic

### 2.1.3 Format Field

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

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

**MIPS32**

Figure 2-4 Example of Instruction Format

The assembler format is shown with literal parts of the assembler instruction printed in uppercase characters. The variable parts, the operands, are shown as the lowercase names of the appropriate fields. The architectural level at which the instruction was first defined, for example “MIPS32” is shown at the right side of the page.

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

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

### 2.1.4 Purpose Field

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

---

**Purpose:**

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

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

---

### 2.1.5 Description Field

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

---

**Description:**  $rd \leftarrow rs + rt$

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs
- If the addition does not overflow, the 32-bit result is placed into GPR *rd*

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

---

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

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

### 2.1.6 Restrictions Field

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

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

**Restrictions:**

None

**Figure 2-7 Example of Instruction Restrictions****2.1.7 Operation Field**

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

**Operation:**

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

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

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

**2.1.8 Exceptions Field**

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

**Exceptions:**

Integer Overflow

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

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

**2.1.9 Programming Notes and Implementation Notes Fields**

---

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

---

**Programming Notes:**

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

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

---

## 2.2 Operation Section Notation and Functions

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

This section presents information about the following topics:

- “Instruction Execution Ordering” on page 12
- “Pseudocode Functions” on page 12

### 2.2.1 Instruction Execution Ordering

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

### 2.2.2 Pseudocode Functions

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

- “Coprocessor General Register Access Functions” on page 12
- “Load Memory and Store Memory Functions” on page 14
- “Access Functions for Floating Point Registers” on page 16
- “Miscellaneous Functions” on page 17

#### 2.2.2.1 Coprocessor General Register Access Functions

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

##### *COP\_LW*

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

```

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

  /* Coprocessor-dependent action */

endfunction COP_LW

```

**Figure 2-11 COP\_LW Pseudocode Function*****COP\_LD***

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

```

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

  /* Coprocessor-dependent action */

endfunction COP_LD

```

**Figure 2-12 COP\_LD Pseudocode Function*****COP\_SW***

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

```

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

  /* Coprocessor-dependent action */

endfunction COP_SW

```

**Figure 2-13 COP\_SW Pseudocode Function*****COP\_SD***

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

```

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

  /* Coprocessor-dependent action */

```

```
endfunction COP_SD
```

**Figure 2-14 COP\_SD Pseudocode Function****2.2.2.2 Load Memory and Store Memory Functions**

Regardless of byte ordering (big- or little-endian), the address of a halfword, word, or doubleword is the smallest byte address of the bytes that form the object. For big-endian ordering this is the most-significant byte; for a little-endian ordering this is the least-significant byte.

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

***AddressTranslation***

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

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

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

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

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

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

endfunction AddressTranslation
```

**Figure 2-15 AddressTranslation Pseudocode Function*****LoadMemory***

The LoadMemory function loads a value from memory.

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

```

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

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

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

endfunction LoadMemory

```

**Figure 2-16 LoadMemory Pseudocode Function*****StoreMemory***

The StoreMemory function stores a value to memory.

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

```

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

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

endfunction StoreMemory

```

**Figure 2-17 StoreMemory Pseudocode Function*****Prefetch***

The Prefetch function prefetches data from memory.

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

```

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

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

```

```

/* vAddr: virtual address */
/* DATA: Indicates that access is for DATA */
/* hint: hint that indicates the possible use of the data */

endfunction Prefetch

```

**Figure 2-18 Prefetch Pseudocode Function**

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

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

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

### 2.2.2.3 Access Functions for Floating Point Registers

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

#### *ValueFPR*

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

```

value ← ValueFPR(fpr, fmt)

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

/* fpr: The FPR number */
/* fmt: The format of the data, one of: */
/*      S, D, W, */
/*      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 ← FPR[fpr]

  D, UNINTERPRETED_DOUBLEWORD:
    if (fpr0 ≠ 0) then
      valueFPR ← UNPREDICTABLE
    else

```



```

        valueFPR ← FPR[fpr+1] || FPR[fpr]
    endif

    DEFAULT:
        valueFPR ← UNPREDICTABLE

endcase
endfunction ValueFPR

```

**Figure 2-19 ValueFPR Pseudocode Function*****StoreFPR***

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

```

StoreFPR (fpr, fmt, value)

/* fpr:   The FPR number */
/* fmt:   The format of the data, one of: */
/*        S, D, W, */
/*        OB, QH, */
/*        UNINTERPRETED_WORD, */
/*        UNINTERPRETED_DOUBLEWORD */
/* value: The formatted 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] ← value

    D, UNINTERPRETED_DOUBLEWORD:
        if (fpr0 ≠ 0) then
            UNPREDICTABLE
        else
            FPR[fpr] ← value
            FPR[fpr+1] ← value
        endif

endcase

endfunction StoreFPR

```

**Figure 2-20 StoreFPR Pseudocode Function****2.2.2.4 Miscellaneous Functions**

This section lists miscellaneous functions not covered in previous sections.

***SyncOperation***

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

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

```
SyncOperation(stype)

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

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

endfunction SyncOperation
```

**Figure 2-21 SyncOperation Pseudocode Function**

### ***SignalException***

The SignalException function signals an exception condition.

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

```
SignalException(Exception, argument)

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

endfunction SignalException
```

**Figure 2-22 SignalException Pseudocode Function**

### ***SignalDebugBreakpointException***

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

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

```
SignalDebugBreakpointException()

endfunction SignalDebugBreakpointException
```

**Figure 2-23 SignalDebugBreakpointException Pseudocode Function**

### ***SignalDebugModeBreakpointException***

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

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

```
SignalDebugModeBreakpointException()

endfunction SignalDebugModeBreakpointException
```

**Figure 2-24 SignalDebugModeBreakpointException Pseudocode Function**

***NullifyCurrentInstruction***

The NullifyCurrentInstruction function nullifies the current instruction.

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

```
NullifyCurrentInstruction()

endfunction NullifyCurrentInstruction
```

**Figure 2-25 NullifyCurrentInstruction PseudoCode Function**

***CoprocessorOperation***

The CoprocessorOperation function performs the specified Coprocessor operation.

```
CoprocessorOperation (z, cop_fun)

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

/* Transmit the cop_fun value to coprocessor z */

endfunction CoprocessorOperation
```

**Figure 2-26 CoprocessorOperation Pseudocode Function**

***JumpDelaySlot***

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

```
JumpDelaySlot(vAddr)

/* vAddr:Virtual address */

endfunction JumpDelaySlot
```

**Figure 2-27 JumpDelaySlot Pseudocode Function**

***FPCConditionCode***

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

```
tf ← FPCConditionCode(cc)

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

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

if cc = 0 then
    FPCConditionCode ← FCSR23
else
    FPCConditionCode ← FCSR24+cc
endif
```

```
endfunction FPConditionCode
```

**Figure 2-28 FPConditionCode Pseudocode Function*****SetFPConditionCode***

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

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

**Figure 2-29 SetFPConditionCode Pseudocode Function****2.3 Op and Function Subfield Notation**

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

**2.4 FPU Instructions**

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

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

Bit encodings for mnemonics are given in Volume I, in the chapters describing the CPU, FPU, MDMX, and MIPS16e instructions.

See Section 2.3, "Op and Function Subfield Notation" on page 20 for a description of the *op* and *function* subfields.

# The MIPS32™ Instruction Set

## 3.1 Compliance and Subsetting

To be compliant with the MIPS32 Architecture, designs must implement a set of required features, as described in this document set. To allow flexibility in implementations, the MIPS32 Architecture does provide subsetting rules. An implementation that follows these rules is compliant with the MIPS32 Architecture as long as it adheres strictly to the rules, and fully implements the remaining instructions. Supersetting of the MIPS32 Architecture is only allowed by adding functions to the *SPECIAL2* major opcode, by adding control for co-processors via the *COP2*, *LWC2*, *SWC2*, *LDC2*, and/or *SDC2*, and/or *COP3* opcodes, or via the addition of approved Application Specific Extensions. Note, however, that a decision to use the *COP3* opcode in an implementation of the MIPS32 Architecture precludes a compatible upgrade to the MIPS64 Architecture because the *COP3* opcode is used as part of the floating point ISA in the MIPS64 Architecture.

The instruction set subsetting rules are as follows:

- All CPU instructions must be implemented - no subsetting is allowed.
- The FPU and related support instructions, including the *MOVF* and *MOVT* CPU instructions, may be omitted. Software may determine if an FPU is implemented by checking the state of the *FP* bit in the *Config1* CP0 register. If the FPU is implemented, it must include S, D, and W formats, operate instructions, and all supporting instructions. Software may determine which FPU data types are implemented by checking the appropriate bit in the *FIR* CP1 register. The following allowable FPU subsets are compliant with the MIPS32 architecture:
  - No FPU
  - FPU with S, D, and W formats and all supporting instructions
- Coprocessor 2 is optional and may be omitted. Software may determine if Coprocessor 2 is implemented by checking the state of the *C2* bit in the *Config1* CP0 register. If Coprocessor 2 is implemented, the Coprocessor 2 interface instructions (*BC2*, *CFC2*, *COP2*, *CTC2*, *LDC2*, *LWC2*, *MFC2*, *MTC2*, *SDC2*, and *SWC2*) may be omitted on an instruction-by-instruction basis.
- Supervisor Mode is optional. If Supervisor Mode is not implemented, bit 3 of the *Status* register must be ignored on write and read as zero.
- The standard TLB-based memory management unit may be replaced with a simpler MMU (e.g., a Fixed Mapping MMU). If this is done, the rest of the interface to the Privileged Resource Architecture must be preserved. If a TLB-based memory management unit is implemented, it must be the standard TLB-based MMU as described in the Privileged Resource Architecture chapter. Software may determine the type of the MMU by checking the *MT* field in the *Config* CP0 register.
- The Privileged Resource Architecture includes several implementation options and may be subsetting in accordance with those options.
- Instruction, CP0 Register, and CP1 Control Register fields that are marked “Reserved” or shown as “0” in the description of that field are reserved for future use by the architecture and are not available to implementations. Implementations may only use those fields that are explicitly reserved for implementation dependent use.
- Supported ASEs are optional and may be subsetting out. In most cases, software may determine if a supported ASE is implemented by checking the appropriate bit in the *Config1* or *Config3* CP0 register. If they are implemented, they must implement the entire ISA applicable to the component, or implement subsets that are approved by the ASE specifications.
- EJTAG is optional and may be subsetting out. If it is implemented, it must implement only those subsets that are approved by the EJTAG specification.

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

## 3.2 Alphabetical List of Instructions

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

**Table 3-1 CPU Arithmetic Instructions**

Mnemonic	Instruction
ADD	Add Word
ADDI	Add Immediate Word
ADDIU	Add Immediate Unsigned Word
ADDU	Add Unsigned Word
CLO	Count Leading Ones in Word
CLZ	Count Leading Zeros in Word
DIV	Divide Word
DIVU	Divide Unsigned Word
MADD	Multiply and Add Word to Hi, Lo
MADDU	Multiply and Add Unsigned Word to Hi, Lo
MSUB	Multiply and Subtract Word to Hi, Lo
MSUBU	Multiply and Subtract Unsigned Word to Hi, Lo
MUL	Multiply Word to GPR
MULT	Multiply Word
MULTU	Multiply Unsigned Word
SLT	Set on Less Than
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
SLTU	Set on Less Than Unsigned
SUB	Subtract Word
SUBU	Subtract Unsigned Word

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

Mnemonic	Instruction
B	Unconditional Branch
BAL	Branch and Link
BEQ	Branch on Equal

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

<b>Mnemonic</b>	<b>Instruction</b>
BGEZ	Branch on Greater Than or Equal to Zero
BGEZAL	Branch on Greater Than or Equal to Zero and Link
BGTZ	Branch on Greater Than Zero
BLEZ	Branch on Less Than or Equal to Zero
BLTZ	Branch on Less Than Zero
BLTZAL	Branch on Less Than Zero and Link
BNE	Branch on Not Equal
J	Jump
JAL	Jump and Link
JALR	Jump and Link Register
JR	Jump Register

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

<b>Mnemonic</b>	<b>Instruction</b>
NOP	No Operation
SSNOP	Superscalar No Operation

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

<b>Mnemonic</b>	<b>Instruction</b>
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LL	Load Linked Word
LW	Load Word
LWL	Load Word Left
LWR	Load Word Right
PREF	Prefetch
SB	Store Byte
SC	Store Conditional Word
SD	Store Doubleword
SH	Store Halfword
SW	Store Word

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

<b>Mnemonic</b>	<b>Instruction</b>
SWL	Store Word Left
SWR	Store Word Right
SYNC	Synchronize Shared Memory

**Table 3-5 CPU Logical Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
AND	And
ANDI	And Immediate
LUI	Load Upper Immediate
NOR	Not Or
OR	Or
ORI	Or Immediate
XOR	Exclusive Or
XORI	Exclusive Or Immediate

**Table 3-6 CPU Move Instructions**

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

**Table 3-7 CPU Shift Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
SLL	Shift Word Left Logical
SLLV	Shift Word Left Logical Variable
SRA	Shift Word Right Arithmetic
SRAV	Shift Word Right Arithmetic Variable
SRL	Shift Word Right Logical



**Table 3-7 CPU Shift Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
SRLV	Shift Word Right Logical Variable

**Table 3-8 CPU Trap Instructions**

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

**Table 3-9 Obsolete<sup>1</sup> CPU Branch Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
BEQL	Branch on Equal Likely
BGEZALL	Branch on Greater Than or Equal to Zero and Link Likely
BGEZL	Branch on Greater Than or Equal to Zero Likely
BGTZL	Branch on Greater Than Zero Likely
BLEZL	Branch on Less Than or Equal to Zero Likely
BLTZALL	Branch on Less Than Zero and Link Likely
BLTZL	Branch on Less Than Zero Likely
BNEL	Branch on Not Equal Likely

1. Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS32 architecture.

**Table 3-10 FPU Arithmetic Instructions**

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

**Table 3-11 FPU Branch Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
BC1F	Branch on FP False
BC1T	Branch on FP True

**Table 3-12 FPU Compare Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
C.cond.fmt	Floating Point Compare

**Table 3-13 FPU Convert Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
CEIL.W.fmt	Floating Point Ceiling Convert to Word Fixed Point
CVT.D.fmt	Floating Point Convert to Double Floating Point
CVT.S.fmt	Floating Point Convert to Single Floating Point
CVT.W.fmt	Floating Point Convert to Word Fixed Point
FLOOR.W.fmt	Floating Point Floor Convert to Word Fixed Point
ROUND.W.fmt	Floating Point Round to Word Fixed Point
TRUNC.W.fmt	Floating Point Truncate to Word Fixed Point

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

<b>Mnemonic</b>	<b>Instruction</b>
LDC1	Load Doubleword to Floating Point
LWC1	Load Word to Floating Point
SDC1	Store Doubleword from Floating Point
SWC1	Store Word from Floating Point

**Table 3-15 FPU Move Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
CFC1	Move Control Word from Floating Point
CTC1	Move Control Word to Floating Point
MFC1	Move Word from Floating Point
MOV.fmt	Floating Point Move
MOVF.fmt	Floating Point Move Conditional on Floating Point False
MOVN.fmt	Floating Point Move Conditional on Not Zero
MOVT.fmt	Floating Point Move Conditional on Floating Point True
MOVZ.fmt	Floating Point Move Conditional on Zero
MTC1	Move Word to Floating Point

**Table 3-16 Obsolete<sup>1</sup> FPU Branch Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
BC1FL	Branch on FP False Likely
BC1TL	Branch on FP True Likely

1. Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS32 architecture.

**Table 3-17 Coprocessor Branch Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
BC2F	Branch on COP2 False
BC2T	Branch on COP2 True

**Table 3-18 Coprocessor Execute Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
COP2	Coprocessor Operation to Coprocessor 2

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

<b>Mnemonic</b>	<b>Instruction</b>
LDC2	Load Doubleword to Coprocessor 2
LWC2	Load Word to Coprocessor 2
SDC2	Store Doubleword from Coprocessor 2
SWC2	Store Word from Coprocessor 2

**Table 3-20 Coprocessor Move Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
CFC2	Move Control Word from Coprocessor 2
CTC2	Move Control Word to Coprocessor 2
MFC2	Move Word from Coprocessor 2
MTC2	Move Word to Coprocessor 2

**Table 3-21 Obsolete<sup>1</sup> Coprocessor Branch Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
BC2FL	Branch on COP2 False Likely
BC2TL	Branch on COP2 True Likely

1. Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS32 architecture.

**Table 3-22 Privileged Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
CACHE	Perform Cache Operation
ERET	Exception Return
MFC0	Move from Coprocessor 0
MTC0	Move to Coprocessor 0
TLBP	Probe TLB for Matching Entry
TLBR	Read Indexed TLB Entry
TLBWI	Write Indexed TLB Entry
TLBWR	Write Random TLB Entry
WAIT	Enter Standby Mode

**Table 3-23 EJTAG Instructions**

<b>Mnemonic</b>	<b>Instruction</b>
DERET	Debug Exception Return
SDBBP	Software Debug Breakpoint

31	26	25	21	20	16	15	11	10	6	5	0
COP1	fmt		0		fs		fd		ABS		
010001			00000						000101		
6	5		5		5		5		6		

**Format:** ABS.S fd, fs  
ABS.D fd, fs

**MIPS32**  
**MIPS32**

**Purpose:**

To compute the absolute value of an FP value

**Description:**  $fd \leftarrow \text{abs}(fs)$

The absolute value of the value in FPR *fs* is placed in FPR *fd*. The operand and result are values in format *fmt*. *Cause* bits are ORed into the *Flag* bits if no exception is taken.

This operation is arithmetic; a NaN operand signals invalid operation.

**Restrictions:**

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

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

**Operation:**

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

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation

**Add Word****ADD**

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

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

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

**Description:**  $rd \leftarrow rs + rt$ 

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR *rd*.

**Restrictions:**

None

**Operation:**

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

**Exceptions:**

Integer Overflow

**Programming Notes:**

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

## Floating Point Add

ADD.fmt

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

**Format:** ADD.S fd, fs, ft  
ADD.D fd, fs, ft

**MIPS32**  
**MIPS32**

### Purpose:

To add floating point values

**Description:**  $fd \leftarrow fs + ft$

The value in FPR *ft* is added to the value in FPR *fs*. The result is calculated to infinite precision, rounded by using to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. *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**.

### Operation:

StoreFPR (fd, fmt, ValueFPR(fs, fmt) +<sub>fmt</sub> ValueFPR(ft, fmt))

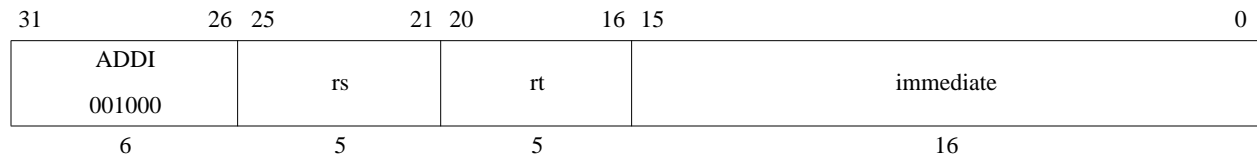
### Exceptions:

Coprocessor Unusable, Reserved Instruction

### Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow





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

**MIPS32**

**Purpose:**

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

**Description:**  $rt \leftarrow rs + \text{immediate}$

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 placed into GPR *rt*.

**Restrictions:**

None

**Operation:**

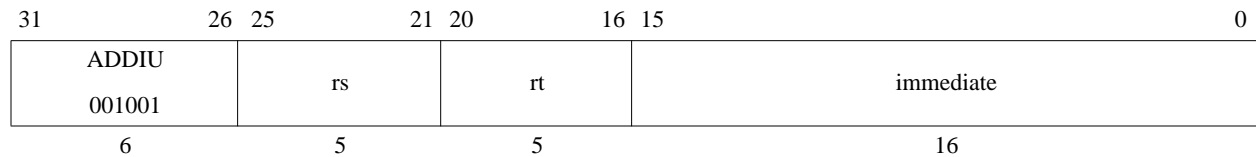
```
temp ← (GPR[rs]31 || GPR[rs]31..0) + sign_extend(immediate)
if temp32 ≠ temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rt] ← temp
endif
```

**Exceptions:**

Integer Overflow

**Programming Notes:**

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



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

**MIPS32**

**Purpose:**

To add a constant to a 32-bit integer

**Description:**  $rt \leftarrow rs + \text{immediate}$

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

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**

None

**Operation:**

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

---

**Add Unsigned Word****ADDU**

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

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

To add 32-bit integers

**Description:**  $rd \leftarrow rs + rt$ 

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

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**

None

**Operation:**

```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

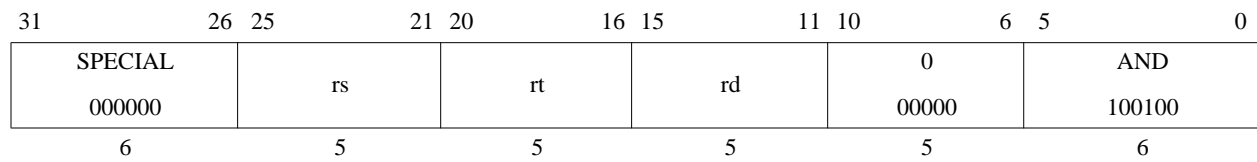
**Exceptions:**

None

**Programming Notes:**

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

---

**And****AND**

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

**MIPS32**

**Purpose:**

To do a bitwise logical AND

**Description:**  $rd \leftarrow rs \text{ AND } rt$

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

**Restrictions:**

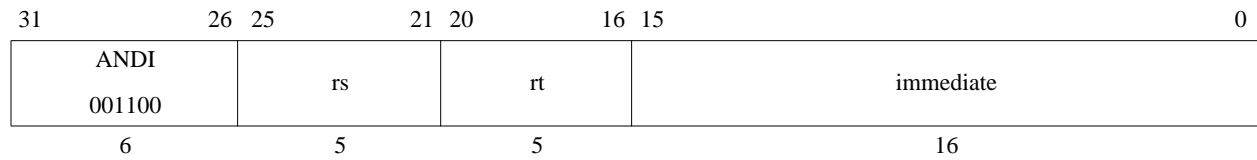
None

**Operation:**

$GPR[rd] \leftarrow GPR[rs] \text{ and } GPR[rt]$

**Exceptions:**

None



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

**MIPS32**

**Purpose:**

To do a bitwise logical AND with a constant

**Description:**  $rt \leftarrow rs \text{ AND } \text{immediate}$

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:**

$GPR[rt] \leftarrow GPR[rs] \text{ and } \text{zero\_extend}(\text{immediate})$

**Exceptions:**

None

## Unconditional Branch

**B**

31	26	25	21	20	16	15	0
BEQ		0		0		offset	
000100		00000		00000			
6		5		5		16	

**Format:** B offset**Assembly Idiom****Purpose:**

To do an unconditional branch

**Description:** branch

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

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

**Restrictions:**

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

**Operation:**

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

**Exceptions:**

None

**Programming Notes:**

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

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

**Format:** BAL *rs*, *offset*

**Assembly Idiom**

**Purpose:**

To do an unconditional PC-relative procedure call

**Description:** `procedure_call`

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

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

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

**Restrictions:**

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

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

**Operation:**

```

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

```

**Exceptions:**

None

**Programming Notes:**

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

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

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

**MIPS32**  
**MIPS32**

#### Purpose:

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

**Description:** if cc = 0 then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the FP condition code bit *CC* is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. An FP condition code is set by the FP compare instruction, C.cond.fmt.

#### Restrictions:

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

#### Operation:

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

```

I:      condition ← FPConditionCode(cc) = 0
          target_offset ← (offset15)GPRLLEN-(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  $\pm 128$  KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range

**Historical Information:**

The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (*Cp1Cond*) and the *C* bit in the FP *Control/Status* register. MIPS I, II, and III architectures must have the *CC* field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more *Condition Code* bits to the original condition code 0. FP compare and conditional branch instructions specify the *Condition Code* bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

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

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

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

**MIPS32**  
**MIPS32**

#### Purpose:

To test an FP condition code and make a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

**Description:** if cc = 0 then branch\_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the FP *Condition Code* bit *CC* is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

An FP condition code is set by the FP compare instruction, C.cond.fmt.

#### Restrictions:

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

#### Operation:

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

```

I:      condition ← FPConditionCode(cc) = 0
          target_offset ← (offset15)GPRLLEN-(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  $\pm 128$  KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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

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

**Historical Information:**

The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (*Cp1Cond*) and the *C* bit in the FP *Control/Status* register. MIPS I, II, and III architectures must have the *CC* field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more *Condition Code* bits to the original condition code 0. FP compare and conditional branch instructions specify the *Condition Code* bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

In the MIPS II 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.

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

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

**MIPS32**  
**MIPS32**

#### Purpose:

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

**Description:** if cc = 1 then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the FP condition code bit *CC* is true (1), the program branches to the effective target address after the instruction in the delay slot is executed. An FP condition code is set by the FP compare instruction, C.cond.fmt.

#### Restrictions:

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

#### Operation:

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

```

I:      condition ← FPConditionCode(cc) = 1
          target_offset ← (offset15)GPRLLEN-(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  $\pm 128$  KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

**Historical Information:**

The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (*Cp1Cond*) and the *C* bit in the FP *Control/Status* register. MIPS I, II, and III architectures must have the *CC* field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more *Condition Code* bits to the original condition code 0. FP compare and conditional branch instructions specify the *Condition Code* bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

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

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

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

**MIPS32**  
**MIPS32**

#### Purpose:

To test an FP condition code and do a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

**Description:** if cc = 1 then branch\_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the FP *Condition Code* bit *CC* is true (1), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

An FP condition code is set by the FP compare instruction, C.cond.fmt.

#### Restrictions:

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

#### Operation:

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

```

I:      condition ← FPConditionCode(cc) = 1
          target_offset ← (offset15)GPRLLEN-(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  $\pm 128$  KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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

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

**Historical Information:**

The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (*Cp1Cond*) and the *C* bit in the FP *Control/Status* register. MIPS I, II, and III architectures must have the *CC* field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more *Condition Code* bits to the original condition code 0. FP compare and conditional branch instructions specify the *Condition Code* bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

In the MIPS II 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.

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

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

**MIPS32**  
**MIPS32**

### Purpose:

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

**Description:** if cc = 0 then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by *CC* is false (0), the program branches to the effective target address after the instruction in the delay slot is executed.

### Restrictions:

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

### Operation:

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

```

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

```

### Exceptions:

Coprocessor Unusable, Reserved Instruction

### Programming Notes:

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



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

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

**MIPS32**  
**MIPS32**

#### Purpose:

To test a COP2 condition code and make a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

**Description:** if cc = 0 then branch\_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by *CC* is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

#### Restrictions:

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

#### Operation:

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

```

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

```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Programming Notes:**

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

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

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

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

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

**MIPS32**  
**MIPS32**

### Purpose:

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

**Description:** if cc = 1 then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by *CC* is true (1), the program branches to the effective target address after the instruction in the delay slot is executed.

### Restrictions:

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

### Operation:

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

```

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

```

### Exceptions:

Coprocessor Unusable, Reserved Instruction

### Programming Notes:

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

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

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

**MIPS32**  
**MIPS32**

#### Purpose:

To test a COP2 condition code and do a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

**Description:** if cc = 1 then branch\_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by *CC* is true (1), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

#### Restrictions:

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

#### Operation:

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

```

I:      condition ← COP2Condition(cc) = 1
          target_offset ← (offset15)GPRLLEN-(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  $\pm 128$  KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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

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

31	26	25	21	20	16	15	0
BEQ 000100		rs		rt		offset	
6		5		5		16	

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

**MIPS32**

**Purpose:**

To compare GPRs then do a PC-relative conditional branch

**Description:** if *rs* = *rt* then branch

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

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

**Restrictions:**

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

**Operation:**

```

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

```

**Exceptions:**

None

**Programming Notes:**

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

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

---

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

31	26	25	21	20	16	15	0
BEQL 010100		rs		rt		offset	
6		5		5		16	

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

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

**Description:** if rs = rt then branch\_likely

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

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

**Restrictions:**

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

**Operation:**

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

**Exceptions:**

None

**Programming Notes:**

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

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

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

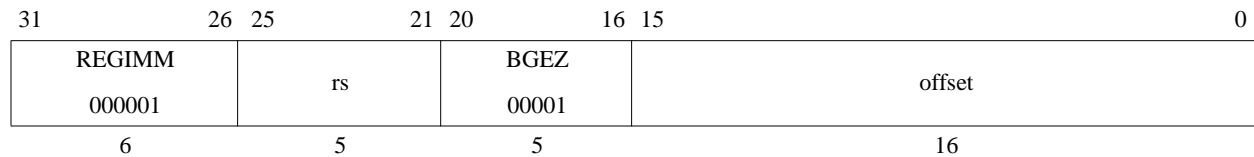
**Historical Information:**

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



## Branch on Greater Than or Equal to Zero

**BGEZ**



**Format:** BGEZ *rs*, *offset*

**MIPS32**

### Purpose:

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

**Description:** if  $rs \geq 0$  then branch

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

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

### Restrictions:

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

### Operation:

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

### Exceptions:

None

### Programming Notes:

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

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

**Format:** BGEZAL *rs*, *offset*

**MIPS32**

**Purpose:**

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

**Description:** if  $rs \geq 0$  then *procedure\_call*

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

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

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

**Restrictions:**

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

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

**Operation:**

```

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

```

**Exceptions:**

None

**Programming Notes:**

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

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

31	26	25	21	20	16	15	0
REGIMM 000001			rs		BGEZALL 10011		offset
6			5		5		16

**Format:** BGEZALL *rs*, *offset*

**MIPS32**

**Purpose:**

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

**Description:** if  $rs \geq 0$  then *procedure\_call\_likely*

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

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

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

**Restrictions:**

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

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

**Operation:**

```

I:    target_offset ← sign_extend(offset || 02)
        condition ← GPR[rs] ≥ 0GPRLEN
        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  $\pm 128$  KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

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

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

**Historical Information:**

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

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

**Format:** BGEZL *rs*, *offset*

**MIPS32**

**Purpose:**

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

**Description:** if  $rs \geq 0$  then *branch\_likely*

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

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

**Restrictions:**

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

**Operation:**

```

I:    target_offset ← sign_extend(offset || 02)
        condition ← GPR[rs] ≥ 0GPRLEN
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  $\pm 128$  KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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

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

**Historical Information:**

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

**Branch on Greater Than Zero****BGTZ**

31	26	25	21	20	16	15	0
BGTZ		rs		0		offset	
000111				00000			
6		5		5		16	

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

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

**Description:** if *rs* > 0 then branch

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

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

**Restrictions:**

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

**Operation:**

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

**Exceptions:**

None

**Programming Notes:**

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

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

**Format:** BGTZL rs, offset

**MIPS32**

**Purpose:**

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

**Description:** if  $rs > 0$  then branch\_likely

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

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

**Restrictions:**

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

**Operation:**

```

I:    target_offset ← sign_extend(offset || 02)
        condition ← GPR[rs] > 0GPRLEN
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  $\pm 128$  KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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

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

**Historical Information:**

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

**Branch on Less Than or Equal to Zero****BLEZ**

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

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

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

**Description:** if  $rs \leq 0$  then branch

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

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

**Restrictions:**

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

**Operation:**

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

**Exceptions:**

None

**Programming Notes:**

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

---

**Branch on Less Than or Equal to Zero Likely****BLEZL**

31	26	25	21	20	16	15	0
BLEZL		rs		0		offset	
010110				00000			
6		5		5		16	

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

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

**Description:** if  $rs \leq 0$  then branch\_likely

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

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

**Restrictions:**

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

**Operation:**

```
I:    target_offset ← sign_extend(offset || 02)
      condition ← GPR[rs] ≤ 0GPRLEN
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  $\pm 128$  KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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

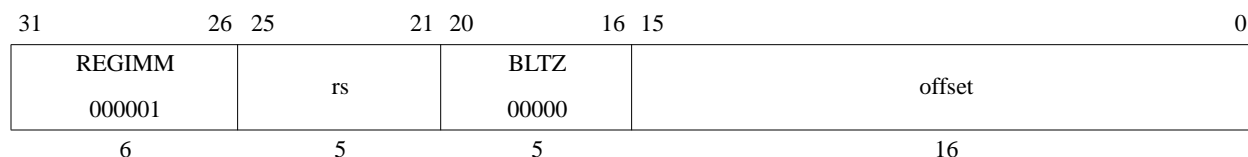
Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability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

**Historical Information:**

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

## Branch on Less Than Zero

BLTZ



**Format:** BLTZ *rs*, *offset*

**MIPS32**

### Purpose:

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

**Description:** if *rs* < 0 then branch

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

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

### Restrictions:

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

### Operation:

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

### Exceptions:

None

### Programming Notes:

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

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

**Format:** BLTZAL *rs*, *offset*

**MIPS32**

**Purpose:**

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

**Description:** if *rs* < 0 then *procedure\_call*

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

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

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

**Restrictions:**

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

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

**Operation:**

```

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

```

**Exceptions:**

None

**Programming Notes:**

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

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

**Format:** BLTZALL *rs*, *offset*

**MIPS32**

**Purpose:**

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

**Description:** if *rs* < 0 then *procedure\_call\_likely*

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

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

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

**Restrictions:**

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

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

**Operation:**

```

I:    target_offset ← sign_extend(offset || 02)
        condition ← GPR[rs] < 0GPRLEN
        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  $\pm 128$  KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

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

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

**Historical Information:**

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



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

**Format:** BLTZL *rs*, *offset*

**MIPS32**

**Purpose:**

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

**Description:** if *rs* < 0 then *branch\_likely*

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

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

**Restrictions:**

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

**Operation:**

```

I:    target_offset ← sign_extend(offset || 02)
        condition ← GPR[rs] < 0GPRLEN
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  $\pm 128$  KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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

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

**Historical Information:**

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

## Branch on Not Equal

**BNE**

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

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

To compare GPRs then do a PC-relative conditional branch

**Description:** if *rs*  $\neq$  *rt* then branch

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

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

**Restrictions:**

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

**Operation:**

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

**Exceptions:**

None

**Programming Notes:**

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

---

**Branch on Not Equal Likely****BNEL**

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

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

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

**Description:** if  $rs \neq rt$  then branch\_likely

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

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

**Restrictions:**

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

**Operation:**

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

**Exceptions:**

None

**Programming Notes:**

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

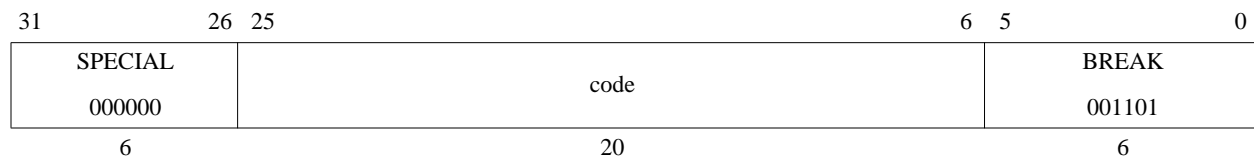
Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

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

**Historical Information:**

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

---

**Breakpoint****BREAK****Format:** BREAK**MIPS32****Purpose:**

To cause a Breakpoint exception

**Description:**

A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler. The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

**Restrictions:**

None

**Operation:**`SignalException(Breakpoint)`**Exceptions:**

Breakpoint

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

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

**MIPS32**  
**MIPS32**  
**MIPS32**  
**MIPS32**

### Purpose:

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

**Description:**  $cc \leftarrow fs \text{ compare\_cond } ft$

The value in FPR *fs* is compared to the value in FPR *ft*; the values are in format *fmt*. The comparison is exact and neither overflows nor underflows.

If the comparison specified by  $cond_{2..1}$  is true for the operand values, the result is true; otherwise, the result is false. If no exception is taken, the result is written into condition code *CC*; true is 1 and false is 0.

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

There are four mutually exclusive ordering relations for comparing floating point values; one relation is always true and the others are false. The familiar relations are *greater than*, *less than*, and *equal*. In addition, the IEEE floating point standard defines the relation *unordered*, which is true when at least one operand value is NaN; NaN compares unordered with everything, including itself. Comparisons ignore the sign of zero, so +0 equals -0.

The comparison condition is a logical predicate, or equation, of the ordering relations such as *less than or equal*, *equal*, *not less than*, or *unordered or equal*. Compare distinguishes among the 16 comparison predicates. The Boolean result of the instruction is obtained by substituting the Boolean value of each ordering relation for the two FP values in the equation. If the *equal* relation is true, for example, then all four example predicates above yield a true result. If the *unordered* relation is true then only the final predicate, *unordered or equal*, yields a true result.

Logical negation of a compare result allows eight distinct comparisons to test for the 16 predicates as shown in . Each mnemonic tests for both a predicate and its logical negation. For each mnemonic, *compare* tests the truth of the first predicate. When the first predicate is true, the result is true as shown in the “If Predicate Is True” column, and the second predicate must be false, and vice versa. (Note that the False predicate is never true and False/True do not follow the normal pattern.)

The truth of the second predicate is the logical negation of the instruction result. After a compare instruction, test for the truth of the first predicate can be made with the Branch on FP True (BC1T) instruction and the truth of the second can be made with Branch on FP False (BC1F).

Table 3-24 shows another set of eight compare operations, distinguished by a *cond*<sub>3</sub> value of 1 and testing the same 16 conditions. For these additional comparisons, if at least one of the operands is a NaN, including Quiet NaN, then an Invalid Operation condition is raised. If the Invalid Operation condition is enabled in the *FCSR*, an Invalid Operation exception occurs.

Table 3-24 FPU Comparisons Without Special Operand Exceptions

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



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

Instruction	Comparison Predicate					Comparison CC Result		Instruction	
Cond Mnemonic	Name of Predicate and Logically Negated Predicate (Abbreviation)	Relation Values				If Predicate Is True	Inv Op Excp If QNaN?	Condition Field	
		>	<	=	?			3	2..0
SF	Signaling False [this predicate always False]	F	F	F	F	F	Yes	1	0
	Signaling True (ST)	T	T	T	T				
NGLE	Not Greater Than or Less Than or Equal	F	F	F	T	T			1
	Greater Than or Less Than or Equal (GLE)	T	T	T	F	F			
SEQ	Signaling Equal	F	F	T	F	T			2
	Signaling Not Equal (SNE)	T	T	F	T	F			
NGL	Not Greater Than or Less Than	F	F	T	T	T			3
	Greater Than or Less Than (GL)	T	T	F	F	F			
LT	Less Than	F	T	F	F	T			4
	Not Less Than (NLT)	T	F	T	T	F			
NGE	Not Greater Than or Equal	F	T	F	T	T			5
	Greater Than or Equal (GE)	T	F	T	F	F			
LE	Less Than or Equal	F	T	T	F	T			6
	Not Less Than or Equal (NLE)	T	F	F	T	F			
NGT	Not Greater Than	F	T	T	T	T			7
	Greater Than (GT)	T	F	F	F	F			
Key: ? = unordered, > = greater than, < = less than, = is equal, T = True, F = False									

**Restrictions:**

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

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

**Operation:**

```

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)

```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation

**Programming Notes:**

FP computational instructions, including compare, that receive an operand value of Signaling NaN raise the Invalid Operation condition. Comparisons that raise the Invalid Operation condition for Quiet NaNs in addition to SNaNs permit a simpler programming model if NaNs are errors. Using these compares, programs do not need explicit code to check for QNaNs causing the *unordered* relation. Instead, they take an exception and allow the exception handling system to deal with the error when it occurs. For example, consider a comparison in which we want to know if two numbers are equal, but for which *unordered* would be an error.

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

**Historical Information:**

The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (*Cp1Cond*) and the *C* bit in the FP *Control/Status* register. MIPS I, II, and III architectures must have the *CC* field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more *Condition Code* bits to the original condition code 0. FP compare and conditional branch instructions specify the *Condition Code* bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

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

## Perform Cache Operation

CACHE

31	26	25	21	20	16	15	0
CACHE 101111	base	op	offset				
6	5	5	16				

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

**MIPS32**

### Purpose:

To perform the cache operation specified by op.

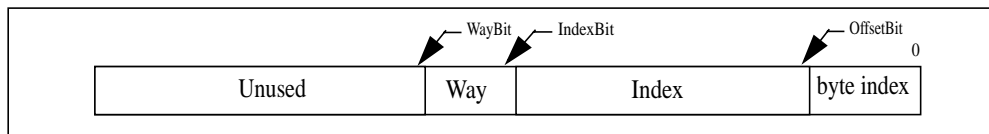
### Description:

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

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

Operation Requires an	Type of Cache	Usage of Effective Address
Address	Virtual	The effective address is used to address the cache. It is implementation dependent whether an address translation is performed on the effective address (with the possibility that a TLB Refill or TLB Invalid exception might occur)
Address	Physical	The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache
Index	N/A	<p>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.</p> <p>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:</p> $\begin{aligned} \text{OffsetBit} &\leftarrow \text{Log2}(\text{BPT}) \\ \text{IndexBit} &\leftarrow \text{Log2}(\text{CS} / \text{A}) \\ \text{WayBit} &\leftarrow \text{IndexBit} + \text{Ceiling}(\text{Log2}(\text{A})) \\ \text{Way} &\leftarrow \text{Addr}_{\text{WayBit}-1..\text{IndexBit}} \\ \text{Index} &\leftarrow \text{Addr}_{\text{IndexBit}-1..\text{OffsetBit}} \end{aligned}$ <p>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.</p>

Figure 3-1 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.

A Cache Error exception may occur as a byproduct of some operations performed by this instruction. For example, if a Writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error. However, cache error exceptions should must be triggered by an Index Load Tag or Index Store tag operation, as these operations are used for initialization and diagnostic purposes.

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

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

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

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

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

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

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

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

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

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance
2#010	All	Index Store Tag	Index	<p>Write the tag for the cache block at the specified index from the <i>TagLo</i> and <i>TagHi</i> Coprocessor 0 registers. This operation must not cause a Cache Error Exception.</p> <p>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 <i>TagLo</i> and <i>TagHi</i> registers associated with the cache be initialized first.</p>	Required
2#011	All	Implementation Dependent	Unspecified	Available for implementation-dependent operation.	Optional
2#100	I, D	Hit Invalidate	Address	<p>If the cache block contains the specified address, set the state of the cache block to invalid.</p> <p>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.</p>	Required (Instruction Cache Encoding Only), Recommended otherwise
	S, T	Hit Invalidate	Address		Optional

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

<b>Code</b>	<b>Caches</b>	<b>Name</b>	<b>Effective Address Operand Type</b>	<b>Operation</b>	<b>Compliance</b>
2#101	I	Fill	Address	Fill the cache from the specified address.	Recommended
	D	Hit Writeback Invalidate / Hit Invalidate	Address	For a write-back cache: If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid.	Required
	S, T	Hit Writeback Invalidate / Hit Invalidate	Address	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.	Optional
2#110	D	Hit Writeback	Address	If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state. For a write-through cache, this operation may be treated as a nop.	Recommended
	S, T	Hit Writeback	Address		Optional



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

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance
2#111	I, D	Fetch and Lock	Address	<p>If the cache does not contain the specified address, fill it from memory, performing a writeback if required, and set the state to valid and locked. If the cache already contains the specified address, set the state to locked. In set-associative or fully-associative caches, the way selected on a fill from memory is implementation dependent.</p> <p>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.</p> <p>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.</p> <p>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.</p>	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 uncachable.

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

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

**Operation:**

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

**Exceptions:**

TLB Refill Exception.

TLB Invalid Exception

Coprocessor Unusable Exception

Address Error Exception

Cache Error Exception

Bus Error Exception

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

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

**MIPS32**  
**MIPS32**

#### Purpose:

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

**Description:**  $fd \leftarrow \text{convert\_and\_round}(fs)$

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

When the source value is Infinity, NaN, or rounds to an integer outside the range  $-2^{31}$  to  $2^{31}-1$ , the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result,  $2^{31}-1$ , is written to *fd*.

#### Restrictions:

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

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

#### Operation:

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

#### Exceptions:

Coprocessor Unusable, Reserved Instruction

#### Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow

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

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

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

**Description:**  $rt \leftarrow FP\_Control[fs]$ Copy the 32-bit word from FP (coprocessor 1) control register *fs* into GPR *rt*.**Restrictions:**There are a few control registers defined for the floating point unit. The result is **UNPREDICTABLE** if *fs* specifies a register that does not exist.**Operation:**

```

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

```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Historical Information:**

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

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

## Move Control Word From Coprocessor 2

CFC2

31	26	25	21	20	16	15	11	10	0
COP2						CF			
010010						00010			
rt						Impl			
6						5			
5						5			
16						16			

**Format:** CFC2 *rt*, *rd*

**MIPS32**

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

### Purpose:

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

**Description:**  $rt \leftarrow CP2CCR[Impl]$

Copy the 32-bit word from 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 *Impl* specifies a register that does not exist.

### Operation:

```
temp ← CP2CCR[Impl]
GPR[rt] ← temp
```

### Exceptions:

Coprocessor Unusable, Reserved Instruction

## Count Leading Ones in Word

CLO

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

**Format:** CLO rd, rs

**MIPS32**

### Purpose:

To Count the number of leading ones in a word

**Description:**  $rd \leftarrow \text{count\_leading\_ones } rs$

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

### Restrictions:

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

### Operation:

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

### Exceptions:

None

## Count Leading Zeros in Word

CLZ

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

**Format:** CLZ rd, rs

**MIPS32**

### Purpose

Count the number of leading zeros in a word

**Description:**  $rd \leftarrow \text{count\_leading\_zeros } rs$

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

### Restrictions:

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

### Operation:

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

### Exceptions:

None



31	26	25	24	0
COP2	CO	cofun		
010010	1			
6	1	25		

**Format:** COP2 func**MIPS32****Purpose:**

To performance an operation to Coprocessor 2

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

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

**Restrictions:****Operation:**

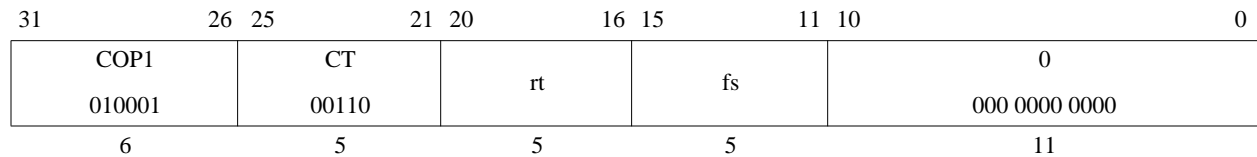
CoprocessorOperation(2, cofun)

**Exceptions:**

Coproprocessor Unusable

Reserved Instruction

---

**Move Control Word to Floating Point****CTC1****Format:** CTC1 rt, fs**MIPS32****Purpose:**

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

**Description:**  $FP\_Control[fs] \leftarrow rt$ Copy the low word from GPR *rt* into the FP (coprocessor 1) control register indicated by *fs*.

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

**Restrictions:**

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

**Operation:**

```

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

```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

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

**Historical Information:**

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

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

## Move Control Word to Coprocessor 2

CTC2

31	26	25	21	20	16	15	11	10	0
COP2						CT		rt	
010010						00110		Impl	
6						5		5	
								16	

**Format:** CTC2 *rt*, *rd*

**MIPS32**

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

### Purpose:

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

**Description:** CP2CCR[Impl] ← *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]
CP2CCR[Impl] ← temp
```

### Exceptions:

Coprocessor Unusable, Reserved Instruction

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

**Format:** CVT.D.S fd, fs  
 CVT.D.W fd, fs  
 CVT.D.L fd, fs

**MIPS32**  
**MIPS32**  
**MIPS64**

#### Purpose:

To convert an FP or fixed point value to double FP

**Description:**  $fd \leftarrow \text{convert\_and\_round}(fs)$

The value in FPR *fs*, in format *fmt*, is converted to a value in double floating point format and rounded according to the current rounding mode in *FCSR*. The result is placed in FPR *fd*. If *fmt* is S or W, then the operation is always exact.

#### Restrictions:

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

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

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

#### Operation:

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

#### Exceptions:

Coprocessor Unusable, Reserved Instruction

#### Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

## Floating Point Convert to Single Floating Point

CVT.S.fmt

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

**Format:** CVT.S.D fd, fs  
 CVT.S.W fd, fs  
 CVT.S.L fd, fs

**MIPS32**  
**MIPS32**  
**MIPS64**

### Purpose:

To convert an FP or fixed point value to single FP

**Description:**  $fd \leftarrow \text{convert\_and\_round}(fs)$

The value in FPR *fs*, in format *fmt*, is converted to a value in single floating point format and rounded according to the current rounding mode in *FCSR*. The result is placed in FPR *fd*.

### Restrictions:

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

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

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

### Operation:

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

### Exceptions:

Coprocessor Unusable, Reserved Instruction

### Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow, Underflow

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

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

**MIPS32**  
**MIPS32**

### Purpose:

To convert an FP value to 32-bit fixed point

**Description:**  $fd \leftarrow \text{convert\_and\_round}(fs)$

The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format and rounded according to the current rounding mode in *FCSR*. The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range  $-2^{31}$  to  $2^{31}-1$ , the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result,  $2^{31}-1$ , is written to *fd*.

### Restrictions:

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

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

### Operation:

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

### Exceptions:

Coprocessor Unusable, Reserved Instruction

### Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow

Debug Exception Return															DERET								
31						26			25	24					6		5	0					
COP0						CO			0										DERET				
010000						1			000 0000 0000 0000 0000										011111				
6						1			19										6				

**Format:** DERET **EJTAG**

**Purpose:**  
To Return from a debug exception.

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

**Restrictions:**  
A DERET placed between an LL and SC instruction does not cause the SC to fail.  
If the DEPC register with the return address for the DERET was modified by an MTC0 or a DMTC0 instruction, a CP0 hazard hazard exists that must be removed via software insertion of the appropriate number of SSNOP instructions.  
The DERET instruction implements a software barrier for all changes in the CP0 state that could affect the fetch and decode of the instruction at the PC to which the DERET returns, such as changes to the effective ASID, user-mode state, and addressing mode.  
This instruction is legal only if the processor is executing in Debug Mode. The operation of the processor is **UNDEFINED** if a DERET is executed in the delay slot of a branch or jump instruction.



**Operation:**

```
DebugDM ← 0
DebugTEXI ← 0
if IsMIPS16Implemented() then
    PC ← DEPC31..1 || 0
    ISAMode ← 0 || DEPC0
else
    PC ← DEPC
endif
```

**Exceptions:**

Coprocessor Unusable Exception  
Reserved Instruction Exception

---

**Divide Word****DIV**

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000			rs		rt		0 00 0000 0000		DIV 011010
6			5		5		10		6

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

To divide a 32-bit signed integers

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

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

No arithmetic exception occurs under any circumstances.

**Restrictions:**

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

**Operation:**

```
q ← GPR[rs]31..0 div GPR[rt]31..0
LO ← q
r ← GPR[rs]31..0 mod GPR[rt]31..0
HI ← r
```

**Exceptions:**

None

**Programming Notes:**

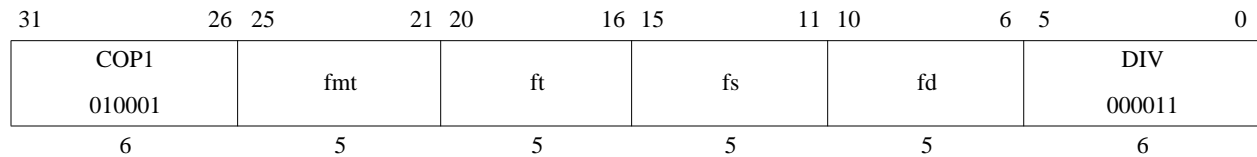
No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions are detected and some action taken, then the divide instruction is typically followed by additional instructions to check for a zero divisor and/or for overflow. If the divide is asynchronous then the zero-divisor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

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

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

**Historical Perspective:**

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



**Format:** DIV.S fd, fs, ft  
DIV.D fd, fs, ft

**MIPS32**  
**MIPS32**

**Purpose:**

To divide FP values

**Description:**  $fd \leftarrow fs / ft$

The value in FPR *fs* is divided by the value in FPR *ft*. The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*.

**Restrictions:**

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

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

**Operation:**

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

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

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

## Divide Unsigned Word

DIVU

31	26	25	21	20	16	15	6	5	0
SPECIAL	rs		rt		0		DIVU		
000000					00 0000 0000		011011		
6	5		5		10		6		

**Format:** DIVU *rs*, *rt*

**MIPS32**

### Purpose:

To divide a 32-bit unsigned integers

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

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

No arithmetic exception occurs under any circumstances.

### Restrictions:

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

### Operation:

```

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

```

### Exceptions:

None

### Programming Notes:

See “Programming Notes” for the DIV instruction.

### Historical Perspective:

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

## Exception Return

ERET

31	26	25	24		6	5	0
COP0	CO	0				ERET	
010000	1	000 0000 0000 0000 0000				011000	
6	1	19				6	

**Format:** ERET

**MIPS32**

### Purpose:

To return from interrupt, exception, or error trap.

### Description:

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

### Restrictions:

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

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

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

### Operation:

```

if StatusERL = 1 then
    temp ← ErrorEPC
    StatusERL ← 0
else
    temp ← EPC
    StatusEXL ← 0
endif
if IsMIPS16Implemented() then
    PC ← temp31..1 || 0
    ISAMode ← temp0
else
    PC ← temp
endif
LLbit ← 0

```

### Exceptions:

Coprocessor Unusable Exception

31	26	25	21	20	16	15	11	10	6	5	0
COP1	fmt		0		fs		fd		FLOOR.W		
010001			00000						001111		
6	5		5		5		5		6		

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

**MIPS32**  
**MIPS32**

### Purpose:

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

**Description:**  $fd \leftarrow \text{convert\_and\_round}(fs)$

The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format and rounded toward  $-\infty$  (rounding mode 3). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range  $-2^{31}$  to  $2^{31}-1$ , the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result,  $2^{31}-1$ , is written to *fd*.

### Restrictions:

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

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

### Operation:

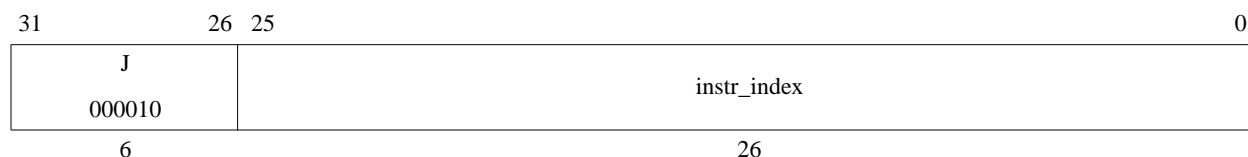
```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

### Exceptions:

Coprocessor Unusable, Reserved Instruction

### Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow



**Format:** J target

**MIPS32**

**Purpose:**

To branch within the current 256 MB-aligned region

**Description:**

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

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

**Restrictions:**

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

**Operation:**

**I:**  

$$I+1:PC \leftarrow PC_{GPREN-1..28} \parallel instr\_index \parallel 0^2$$

**Exceptions:**

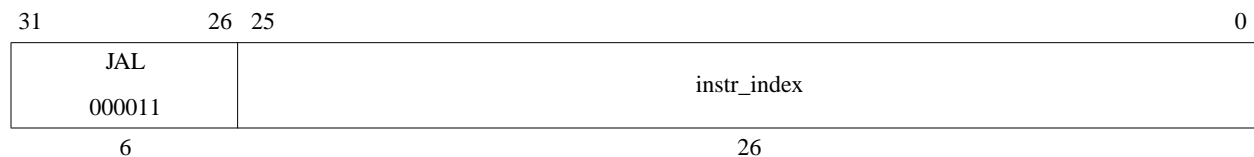
None

**Programming Notes:**

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

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





**Format:** JAL target

**MIPS32**

**Purpose:**

To execute a procedure call within the current 256 MB-aligned region

**Description:**

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call.

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

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

**Restrictions:**

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

**Operation:**

**I:** GPR[31] ← PC + 8  
**I+1:PC** ← PC<sub>GPRLEN-1..28</sub> || instr\_index || 0<sup>2</sup>

**Exceptions:**

None

**Programming Notes:**

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

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

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

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

**MIPS32**  
**MIPS32**

**Purpose:**

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

**Description:** rd ← return\_addr, PC ← rs

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

*For processors that do not implement the MIPS16e ASE:*

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

*For processors that do implement the MIPS16e ASE:*

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

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

**Restrictions:**

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

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16e ASE, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction. For processors that do implement the MIPS16e ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

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

**Operation:**

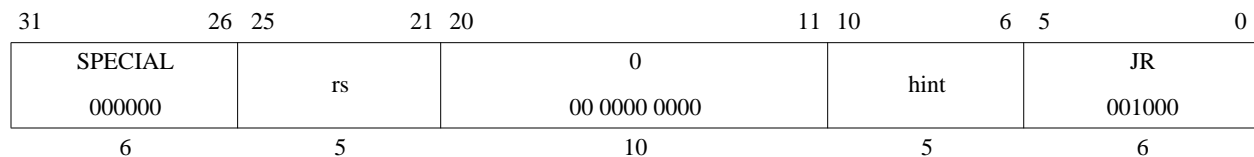
```
I: temp ← GPR[rs]
    GPR[rd] ← PC + 8
I+1: if Config1CA = 0 then
    PC ← temp
    else
    PC ← tempGPRLEN-1..1 || 0
    ISAMode ← temp0
endif
```

**Exceptions:**

None

**Programming Notes:**

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

**Format:** JR *rs***MIPS32****Purpose:**

To execute a branch to an instruction address in a register

**Description:**  $PC \leftarrow rs$ 

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

For processors that implement the MIPS16e ASE, set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

**Restrictions:**

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16e ASE, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction. For processors that do implement the MIPS16e ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

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

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

**Operation:**

```

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

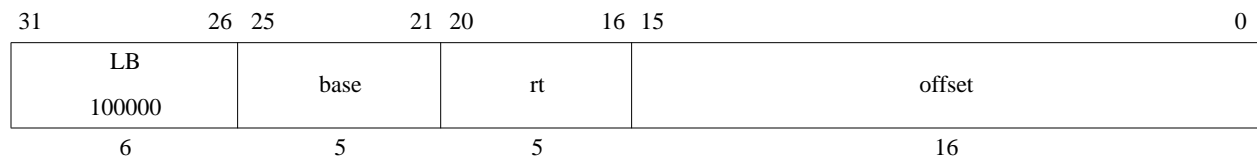
```

**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.



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

**MIPS32**

**Purpose:**

To load a byte from memory as a signed value

**Description:**  $rt \leftarrow \text{memory}[\text{base} + \text{offset}]$

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 ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian2)
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr1..0 xor BigEndianCPU2
GPR[rt] ← sign_extend(memword7+8*byte..8*byte)

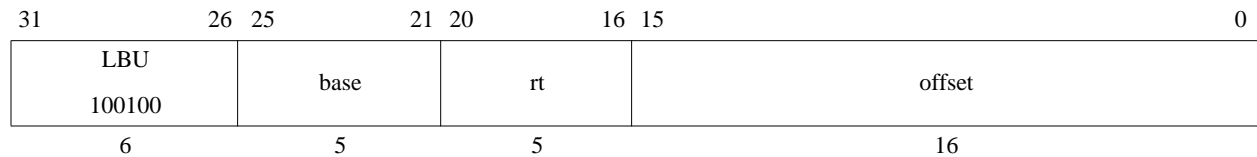
```

**Exceptions:**

TLB Refill, TLB Invalid, Address Error, Watch

## Load Byte Unsigned

**LBU**



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

**MIPS32**

### Purpose:

To load a byte from memory as an unsigned value

**Description:**  $rt \leftarrow \text{memory}[\text{base} + \text{offset}]$

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:

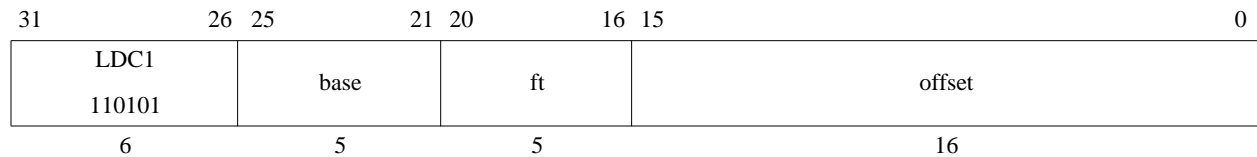
```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian2)
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr1..0 xor BigEndianCPU2
GPR[rt] ← zero_extend(memword7+8*byte..8*byte)

```

### Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch



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

**MIPS32**

**Purpose:**

To load a doubleword from memory to an FPR

**Description:**  $ft \leftarrow \text{memory}[\text{base} + \text{offset}]$

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

**Restrictions:**

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

**Operation:**

```

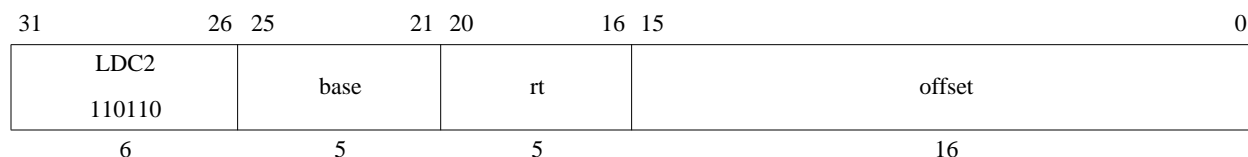
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
paddr ← paddr xor ((BigEndianCPU xor ReverseEndian) || 02)
memlsw ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
paddr ← paddr xor 2#100
memmsw ← LoadMemory(CCA, WORD, pAddr, vAddr+4, DATA)
StoreFPR(ft, UNINTERPRETED_WORD, memlsw)
StoreFPR(ft+1, UNINTERPRETED_WORD, memmsw)

```

**Exceptions:**

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





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

**MIPS32**

**Purpose:**

To load a doubleword from memory to a Coprocessor 2 register

**Description:**  $rt \leftarrow \text{memory}[\text{base} + \text{offset}]$

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in Coprocessor 2 register *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

**Restrictions:**

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

**Operation:**

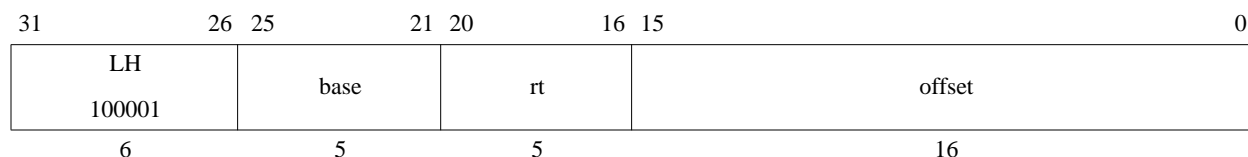
```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then SignalException(AddressError) endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
paddr ← paddr xor ((BigEndianCPU xor ReverseEndian) || 02)
memlsw ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
paddr ← paddr xor 2#100
memmsw ← LoadMemory(CCA, WORD, pAddr, vAddr+4, DATA)
memlsw
memmsw

```

**Exceptions:**

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



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

**MIPS32**

**Purpose:**

To load a halfword from memory as a signed value

**Description:**  $rt \leftarrow \text{memory}[\text{base} + \text{offset}]$

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

**Restrictions:**

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

**Operation:**

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr1..0 xor (BigEndianCPU || 0)
GPR[rt] ← sign_extend(memword15+8*byte..8*byte)

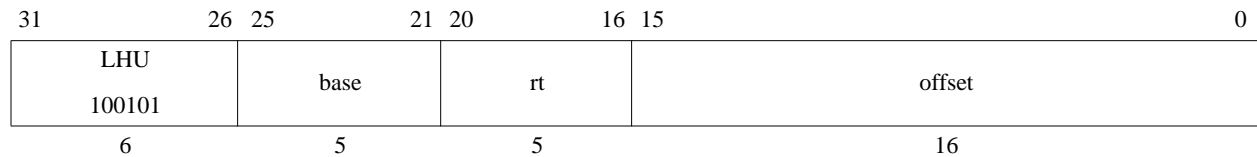
```

**Exceptions:**

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

## Load Halfword Unsigned

LHU



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

**MIPS32**

### Purpose:

To load a halfword from memory as an unsigned value

**Description:**  $rt \leftarrow \text{memory}[\text{base} + \text{offset}]$

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

### Restrictions:

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

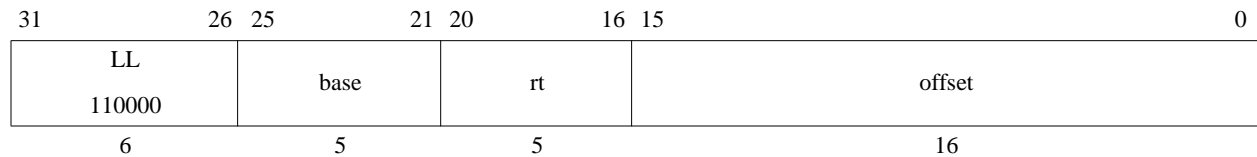
### Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr1..0 xor (BigEndianCPU || 0)
GPR[rt] ← zero_extend(memword15+8*byte..8*byte)
    
```

### Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch



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

**MIPS32**

**Purpose:**

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

**Description:**  $rt \leftarrow \text{memory}[\text{base} + \text{offset}]$

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 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.

**Operation:**

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
LLbit ← 1

```

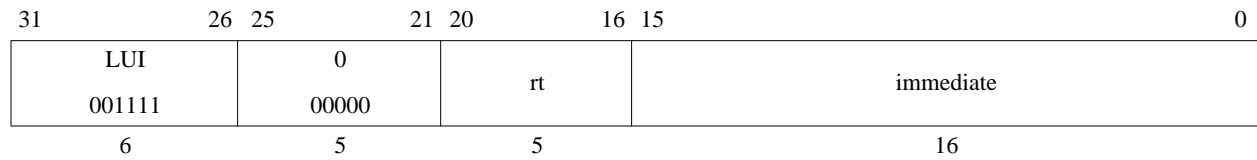
**Exceptions:**

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

**Programming Notes:**

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

---

**Load Upper Immediate****LUI****Format:** LUI *rt*, *immediate***MIPS32****Purpose:**

To load a constant into the upper half of a word

**Description:**  $rt \leftarrow \text{immediate} \parallel 0^{16}$ 

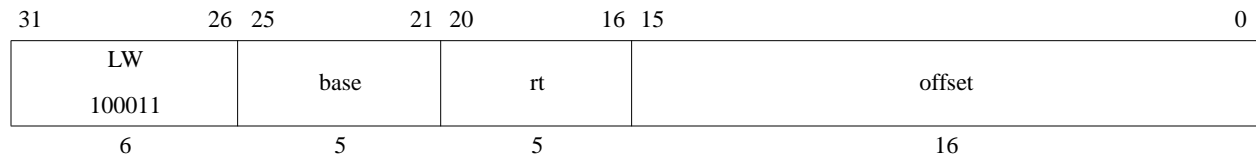
The 16-bit *immediate* is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is placed into GPR *rt*.

**Restrictions:**

None

**Operation:** $\text{GPR}[rt] \leftarrow \text{immediate} \parallel 0^{16}$ **Exceptions:**

None



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

**MIPS32**

**Purpose:**

To load a word from memory as a signed value

**Description:**  $rt \leftarrow \text{memory}[\text{base} + \text{offset}]$

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

**Restrictions:**

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

**Operation:**

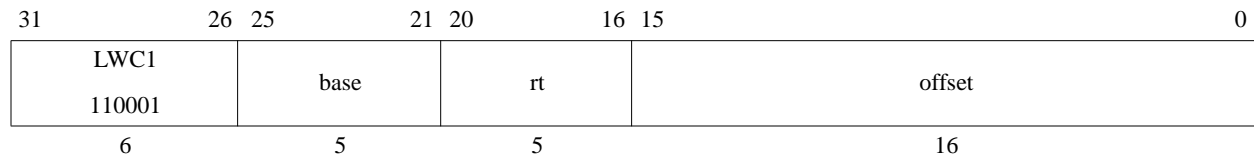
```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword

```

**Exceptions:**

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



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

**MIPS32**

**Purpose:**

To load a word from memory to an FPR

**Description:**  $ft \leftarrow \text{memory}[\text{base} + \text{offset}]$

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of coprocessor 1 general register *ft*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

**Restrictions:**

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

**Operation:**

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)

memword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)

StoreFPR(ft, UNINTERPRETED_WORD,
          memword)

```

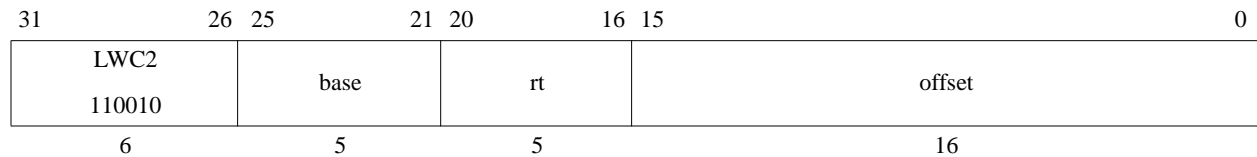
**Exceptions:**

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



## Load Word to Coprocessor 2

LWC2



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

**MIPS32**

### Purpose:

To load a word from memory to a COP2 register

**Description:**  $rt \leftarrow \text{memory}[\text{base} + \text{offset}]$

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of COP2 (Coprocessor 2) general register *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

### Restrictions:

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

### Operation:

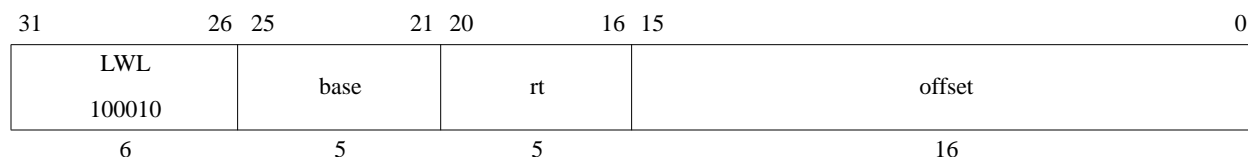
```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr12..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)

memword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)

CPR[2,rt,0] ← memword
```

### Exceptions:

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



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

**MIPS32**

**Purpose:**

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

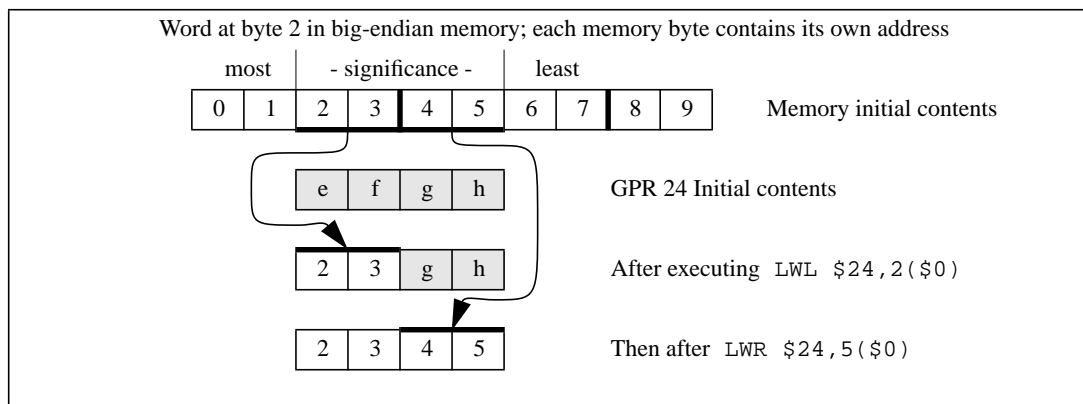
**Description:**  $rt \leftarrow rt \text{ MERGE } \text{memory}[\text{base} + \text{offset}]$

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.

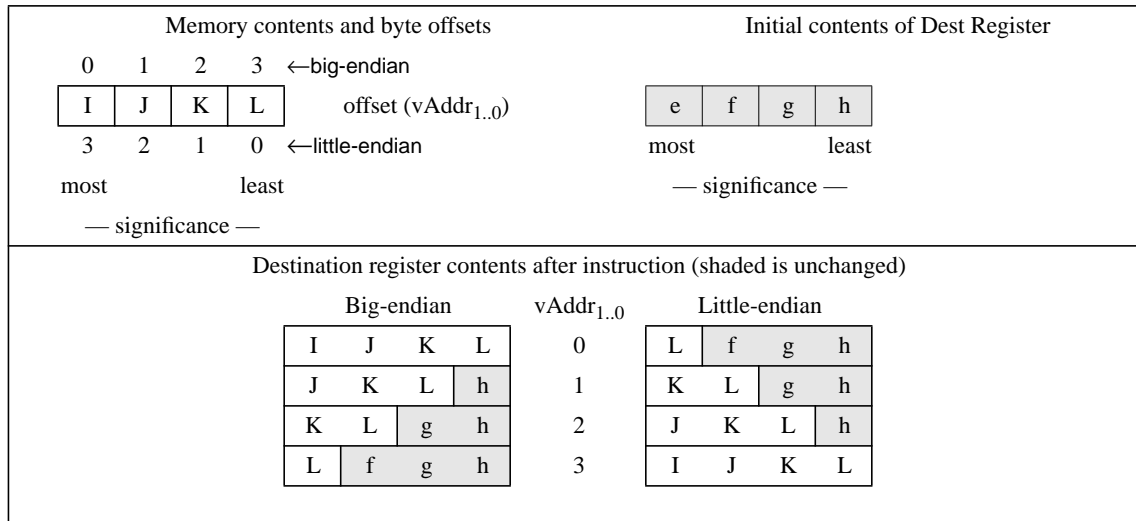
The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2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



**Restrictions:**

None

**Operation:**

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian2)
if BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..2 || 02
endif
byte ← vAddr1..0 xor BigEndianCPU2
memword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp ← memword7+8*byte..0 || GPR[rt]23-8*byte..0
GPR[rt] ← temp

```

**Exceptions:**

None

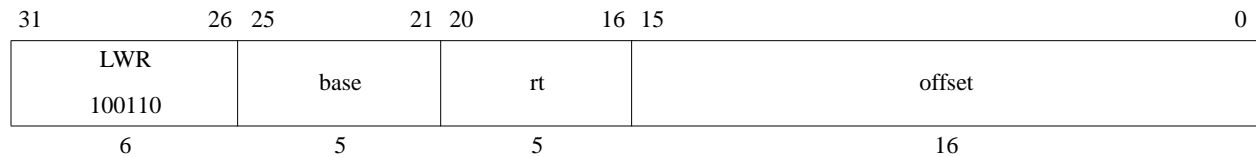
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

**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:** LWR *rt*, offset(*base*)

**MIPS32**

**Purpose:**

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

**Description:**  $rt \leftarrow rt \text{ MERGE } \text{memory}[\text{base} + \text{offset}]$

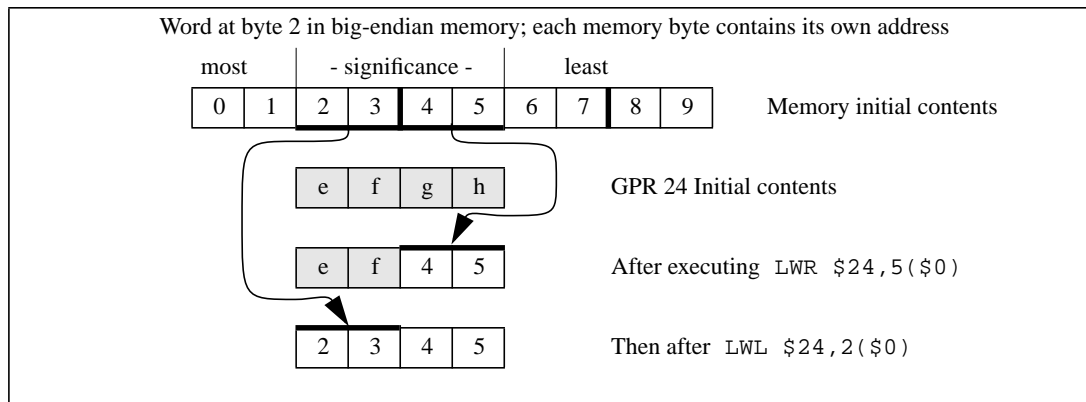
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

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.

Executing both LWR and LWL, in either order, delivers a sign-extended word value in the destination register.

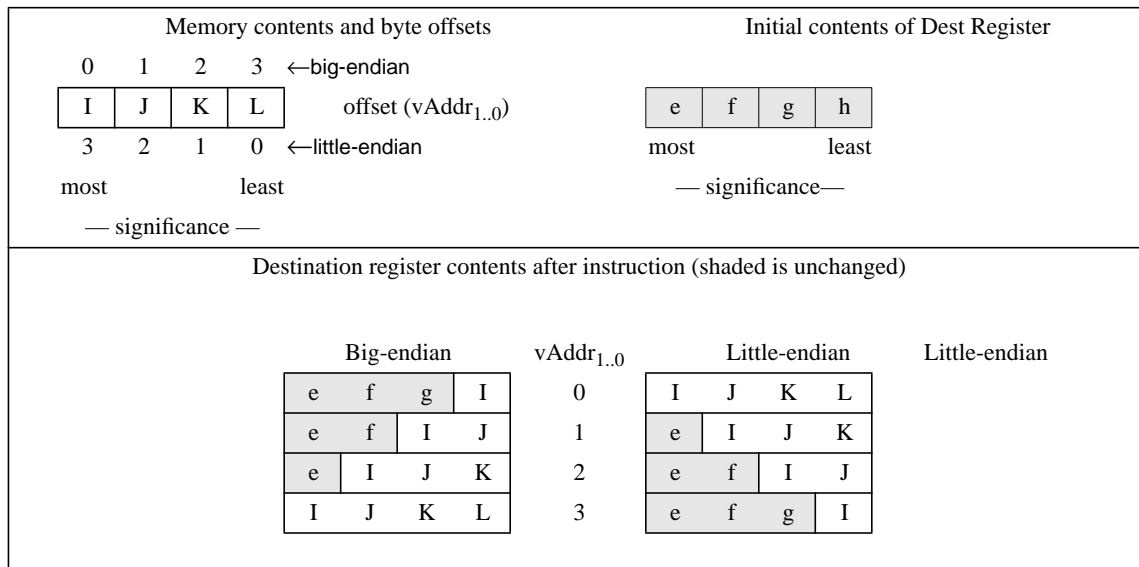
The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the least-significant byte at 5. First, LWR loads these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

Figure 3-4 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 3-5 Bytes Loaded by LWR Instruction



**Restrictions:**

None

**Operation:**

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian2)
if BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..2 || 02
endif
byte ← vAddr1..0 xor BigEndianCPU2
memword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp ← memword31..32-8*byte || GPR[rt]31-8*byte..0
GPR[rt] ← 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.



31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL2			rs		rt		0		0		MADD
011100							0000		00000		000000
6			5		5		5		5		6

**Format:** MADD *rs*, *rt*

**MIPS32**

**Purpose:**

To multiply two words and add the result to Hi, Lo

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

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* and *LO*. The most significant 32 bits of the result are written into *HI* and the least significant 32 bits are written into *LO*. No arithmetic exception occurs under any circumstances.

**Restrictions:**

None

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

```
temp ← (HI || LO) + (GPR[rs] × GPR[rt])
HI ← temp63..32
LO ← temp31..0
```

**Exceptions:**

None

**Programming Notes:**

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

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL2			rs		rt		0		0		MADDU
011100							00000		00000		000001
6			5		5		5		5		6

**Format:** MADDU *rs*, *rt*

**MIPS32**

**Purpose:**

To multiply two unsigned words and add the result to Hi, Lo.

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

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* and *LO*. The most significant 32 bits of the result are written into *HI* and the least significant 32 bits are written into *LO*. No arithmetic exception occurs under any circumstances.

**Restrictions:**

None

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

```
temp ← (HI || LO) + (GPR[rs] × GPR[rt])
HI ← temp63..32
LO ← temp31..0
```

**Exceptions:**

None

**Programming Notes:**

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

**Move from Coprocessor 0****MFC0**

31	26	25	21	20	16	15	11	10	3	2	0
COP0 010000			MF 00000		rt		rd		0 00000000		sel
6			5		5		5		8		3

**Format:** MFC0 rt, rd  
MFC0 rt, rd, sel

**MIPS32**  
**MIPS32**

**Purpose:**

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

**Description:**  $rt \leftarrow CPR[0,rd,sel]$

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

**Restrictions:**

The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *rd* and *sel*.

**Operation:**

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

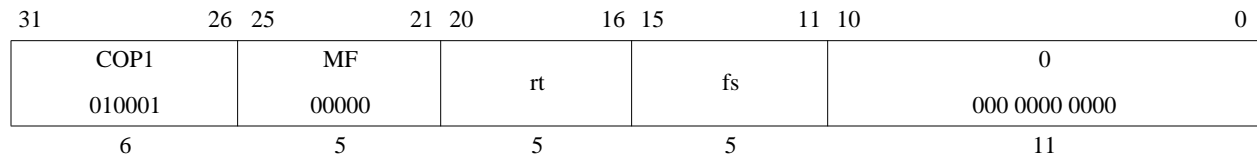
**Exceptions:**

Coprocessor Unusable

Reserved Instruction

## Move Word From Floating Point

MFC1



**Format:** MFC1 rt, fs

**MIPS32**

### Purpose:

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

**Description:**  $rt \leftarrow fs$

The contents of FPR fs are loaded into general register rt.

### Restrictions:

### Operation:

```
data ← ValueFPR(fs, UNINTERPRETED_WORD)
GPR[rt] ← data
```

### Exceptions:

Coprocessor Unusable, Reserved Instruction

### Historical Information:

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

## Move Word From Coprocessor 2

MFC2

31	26	25	21	20	16	15	11	10	8	7	0
COP2						MF					
010010						00000					
rt						Impl					
6						5					

**Format:** MFC2 rt, rd  
MFC2, rt, rd, sel

**MIPS32**  
**MIPS32**

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

### Purpose:

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

**Description:**  $rt \leftarrow CP2CPR[Impl]$

The contents of the coprocessor 2 register denoted by the *Impl* field are 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:

$data \leftarrow CP2CPR[Impl]$   
 $GPR[rt] \leftarrow data$

### Exceptions:

Coprocessor Unusable

---

**Move From HI Register****MFHI**

31	26	25	16	15	11	10	6	5	0
SPECIAL			0			rd	0		MFHI
000000			00 0000 0000				00000		010000
6			10			5	5		6

**Format:** MFHI rd**MIPS32****Purpose:**

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

**Description:**  $rd \leftarrow HI$ 

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

**Restrictions:**

None

**Operation:** $GPR[rd] \leftarrow HI$ **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.

---

**Move From LO Register****MFLO**

31	26	25	16	15	11	10	6	5	0
SPECIAL			0			rd	0		MFLO
000000			00 0000 0000				00000		010010
6			10			5	5		6

**Format:** MFLO rd**MIPS32****Purpose:**

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

**Description:**  $rd \leftarrow LO$ 

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

**Restrictions:** None**Operation:** $GPR[rd] \leftarrow LO$ **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.

31	26	25	21	20	16	15	11	10	6	5	0
COP1	fmt		0		fs		fd		MOV		
010001			00000						000110		
6	5		5		5		5		6		

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

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

**Purpose:**

To move an FP value between FPRs

**Description:**  $fd \leftarrow fs$

The value in FPR *fs* is placed into FPR *fd*. The source and destination are values in format *fmt*.

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

**Restrictions:**

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; 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, ValueFPR(fs, fmt))`

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation



---

**Move Conditional on Floating Point False****MOVF**

31	26	25	21	20	18	17	16	15	11	10	6	5	0
SPECIAL						0	tf	rd				0	MOVCI
000000						0	0					00000	000001
6						1	1	5				5	6

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

To test an FP condition code then conditionally move a GPR

**Description:** `if cc = 0 then rd ← rs`

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

**Restrictions:****Operation:**

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

**Exceptions:**

Reserved Instruction, Coprocessor Unusable

31	26	25	21	20	18	17	16	15	11	10	6	5	0
COP1		fmt			cc		0	tf	fs		fd		MOVCF
010001							0	0					010001
6		5			3		1	1	5		5		6

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

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

**Purpose:**

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

**Description:** if *cc* = 0 then *fd* ← *fs*

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

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

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**.

**Operation:**

```
if FPConditionCode(cc) = 0 then
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation

---

**Move Conditional on Not Zero****MOVN**

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000			rs		rt		rd		0 00000		MOVN 001011
6			5		5		5		5		6

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

To conditionally move a GPR after testing a GPR value

**Description:** if  $rt \neq 0$  then  $rd \leftarrow rs$ If the value in GPR *rt* is not equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.**Restrictions:**

None

**Operation:**

```
if GPR[rt]  $\neq$  0 then
    GPR[rd]  $\leftarrow$  GPR[rs]
endif
```

**Exceptions:**

None

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

31	26	25	21	20	16	15	11	10	6	5	0
COP1		fmt		rt		fs		fd		MOVN	
010001										010011	
6		5		5		5		5		6	

**Format:** MOVN.S fd, fs, rt  
MOVN.D fd, fs, rt

**MIPS32**  
**MIPS32**

**Purpose:**

To test a GPR then conditionally move an FP value

**Description:** if  $rt \neq 0$  then  $fd \leftarrow fs$

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**.

**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

---

**Move Conditional on Floating Point True****MOVT**

31	26	25	21	20	18	17	16	15	11	10	6	5	0
SPECIAL						0	tf	rd				0	MOVCI
000000						0	1					00000	000001
6						1	1	5				5	6

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

To test an FP condition code then conditionally move a GPR

**Description:** if cc = 1 then rd  $\leftarrow$  rsIf the floating point condition code specified by *CC* is one, then the contents of GPR *rs* are placed into GPR *rd*.**Restrictions:****Operation:**

```
if FPConditionCode(cc) = 1 then
    GPR[rd]  $\leftarrow$  GPR[rs]
endif
```

**Exceptions:**

Reserved Instruction, Coprocessor Unusable

31	26	25	21	20	18	17	16	15	11	10	6	5	0
COP1	fmt				cc	0	tf	fs		fd		MOVCF	
010001						0	1					010001	
6	5				3	1	1	5		5		6	

**Format:** MOVT.S fd, fs, cc  
MOVT.D fd, fs, cc

**MIPS32**  
**MIPS32**

**Purpose:**

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

**Description:** if cc = 1 then fd ← fs

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

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

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**.



**Operation:**

```
if FPConditionCode(cc) = 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

---

**Move Conditional on Zero****MOVZ**

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

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

To conditionally move a GPR after testing a GPR value

**Description:** if  $rt = 0$  then  $rd \leftarrow rs$ 

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

**Restrictions:**

None

**Operation:**

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

**Exceptions:**

None

**Programming Notes:**

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

31	26	25	21	20	16	15	11	10	6	5	0
COP1		fmt		rt		fs		fd		MOVZ	
010001										010010	
6		5		5		5		5		6	

**Format:** MOVZ.S fd, fs, rt  
MOVZ.D fd, fs, rt

**MIPS32**  
**MIPS32**

**Purpose:**

To test a GPR then conditionally move an FP value

**Description:** if  $rt = 0$  then  $fd \leftarrow fs$

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**.

**Operation:**

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

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

***Floating Point Exceptions:***

Unimplemented Operation

## Multiply and Subtract Word to Hi,Lo

MSUB

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL2			rs		rt		0		0		MSUB
011100							00000		00000		000100
6			5		5		5		5		6

**Format:** MSUB *rs*, *rt*

**MIPS32**

### Purpose:

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

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

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* and *LO*. The most significant 32 bits of the result are written into *HI* and the least significant 32 bits are written into *LO*. No arithmetic exception occurs under any circumstances.

### Restrictions:

None

This instruction does not provide the capability of writing directly to a target GPR.

### Operation:

```
temp ← (HI || LO) - (GPR[rs] × GPR[rt])
HI ← temp63..32
LO ← temp31..0
```

### Exceptions:

None

### Programming Notes:

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

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL2			rs		rt		0		0		MSUBU
011100							00000		00000		000101
6			5		5		5		5		6

**Format:** MSUBU *rs*, *rt*

**MIPS32**

**Purpose:**

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

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

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* and *LO*. The most significant 32 bits of the result are written into *HI* and the least significant 32 bits are written into *LO*. No arithmetic exception occurs under any circumstances.

**Restrictions:**

None

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

```
temp ← (HI || LO) - (GPR[rs] × GPR[rt])
HI ← temp63..32
LO ← temp31..0
```

**Exceptions:**

None

**Programming Notes:**

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

**Move to Coprocessor 0****MTC0**

31	26	25	21	20	16	15	11	10	3	2	0
COP0 010000			MT 00100		rt		rd		0 0000 000		sel
6			5		5		5		8		3

**Format:** MTC0 rt, rd  
MTC0 rt, rd, sel

**MIPS32**  
**MIPS32**

**Purpose:**

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

**Description:**  $CPR[r0, rd, sel] \leftarrow rt$

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. In those instances, the *sel* field must be set to zero.

**Restrictions:**

The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *rd* and *sel*.

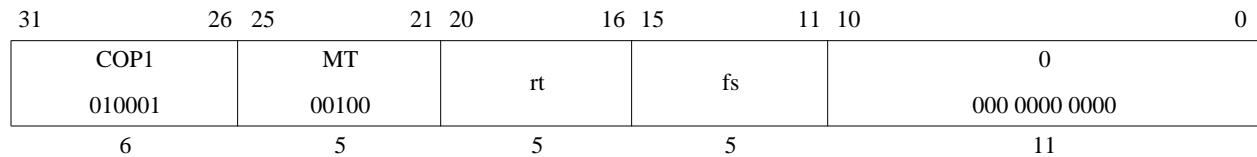
**Operation:**

$CPR[0, rd, sel] \leftarrow data$

**Exceptions:**

Coprocessor Unusable

Reserved Instruction

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

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

**Description:**  $fs \leftarrow rt$ The low word in GPR *rt* is placed into the low word of floating point (Coprocessor 1) general register *fs*.**Restrictions:****Operation:**

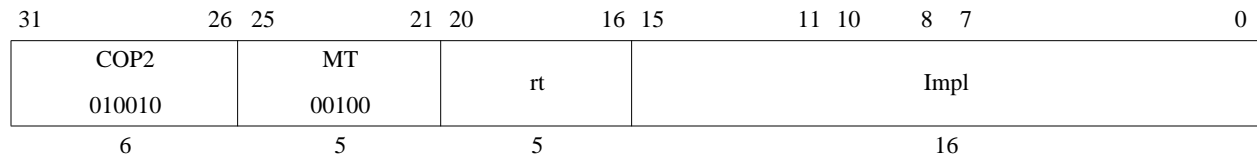
```
data ← GPR[rt]31..0
StoreFPR(fs, UNINTERPRETED_WORD, data)
```

**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, rd  
MTC2 rt, rd, sel

**MIPS32**  
**MIPS32**

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

**Purpose:**

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

**Description:** CP2CPR[Impl]  $\leftarrow$  rt

The low word in GPR *rt* is placed into the low 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.

**Operation:**

data  $\leftarrow$  GPR[rt]  
CP2CPR[Impl]  $\leftarrow$  data

**Exceptions:**

Coprocessor Unusable

Reserved Instruction

31	26	25	21	20	6	5	0
SPECIAL	rs		0			MTHI	
000000			000 0000 0000 0000			010001	
6	5		15			6	

**Format:** MTHI rs

**MIPS32**

**Purpose:**

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

**Description:**  $HI \leftarrow rs$

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

**Restrictions:**

A computed result written to the *HI/LO* pair by DIV, DIVU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either *HI* or *LO*.

If an MTHI instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of *LO* are UNPREDICTABLE. The following example shows this illegal situation:

```

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

```

**Operation:**

$HI \leftarrow GPR[rs]$

**Exceptions:**

None

**Historical Information:**

In MIPS I-III, if either of the two preceding instructions is MFHI, the result of that MFHI is UNPREDICTABLE. Reads of the *HI* or *LO* special register must be separated from any subsequent instructions that write to them by two or more instructions. In MIPS IV and later, including MIPS32 and MIPS64, this restriction does not exist.

31	26	25	21	20	6	5	0
SPECIAL	rs		0			MTLO	
000000			000 0000 0000 0000			010011	
6	5		15			6	

**Format:** MTLO rs

**MIPS32**

**Purpose:**

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

**Description:**  $LO \leftarrow rs$

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

**Restrictions:**

A computed result written to the *HI/LO* pair by DIV, DIVU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either *HI* or *LO*.

If an MTLO instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of *HI* are UNPREDICTABLE. The following example shows this illegal situation:

```

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

```

**Operation:**

$LO \leftarrow GPR[rs]$

**Exceptions:**

None

**Historical Information:**

In MIPS I-III, if either of the two preceding instructions is MFHI, the result of that MFHI is UNPREDICTABLE. Reads of the *HI* or *LO* special register must be separated from any subsequent instructions that write to them by two or more instructions. In MIPS IV and later, including MIPS32 and MIPS64, this restriction does not exist.

---

**Multiply Word to GPR****MUL**

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL2	rs				rt				0		MUL
011100									00000		000010
6	5				5				5		6

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

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

**Description:**  $rd \leftarrow rs \times rt$ 

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 written to GPR *rd*. The contents of *HI* and *LO* are **UNPREDICTABLE** after the operation. No arithmetic exception occurs under any circumstances.

**Restrictions:**

Note that this instruction does not provide the capability of writing the result to the HI and LO registers.

**Operation:**

```
temp <- GPR[rs] * GPR[rt]
GPR[rd] <- temp31..0
HI <- UNPREDICTABLE
LO <- UNPREDICTABLE
```

**Exceptions:**

None

**Programming Notes:**

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

Programs that require overflow detection must check for it explicitly.

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

## Floating Point Multiply

MUL.fmt

31	26	25	21	20	16	15	11	10	6	5	0
COP1	fmt		ft		fs		fd		MUL		
010001									000010		
6	5		5		5		5		6		

**Format:** MUL.S fd, fs, ft  
MUL.D fd, fs, ft

**MIPS32**  
**MIPS32**

### Purpose:

To multiply FP values

**Description:**  $fd \leftarrow fs \times ft$

The value in FPR *fs* is multiplied by the value in FPR *ft*. The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*.

### Restrictions:

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

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

### Operation:

$\text{StoreFPR}(fd, fmt, \text{ValueFPR}(fs, fmt) \times_{\text{fmt}} \text{ValueFPR}(ft, fmt))$

### Exceptions:

Coprocessor Unusable, Reserved Instruction

### Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

## Multiply Word

MULT

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000			rs		rt		0 00 0000 0000		MULT 011000
6			5		5		10		6

**Format:** MULT rs, rt

**MIPS32**

### Purpose:

To multiply 32-bit signed integers

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

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

No arithmetic exception occurs under any circumstances.

### Restrictions:

None

### Operation:

```
prod  ← GPR[rs]31..0 × GPR[rt]31..0
LO    ← prod31..0
HI    ← prod63..32
```

### Exceptions:

None

### Programming Notes:

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

Programs that require overflow detection must check for it explicitly.

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

## Multiply Unsigned Word

MULTU

31	26	25	21	20	16	15	6	5	0
SPECIAL	rs		rt		0		MULTU		
000000					00 0000 0000		011001		
6	5		5		10		6		

**Format:** MULTU rs, rt

**MIPS32**

### Purpose:

To multiply 32-bit unsigned integers

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

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

No arithmetic exception occurs under any circumstances.

### Restrictions:

None

### Operation:

```

prod ← (0 || GPR[rs]31..0) × (0 || GPR[rt]31..0)
LO ← prod31..0
HI ← prod63..32

```

### Exceptions:

None

### Programming Notes:

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

Programs that require overflow detection must check for it explicitly.

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

31	26	25	21	20	16	15	11	10	6	5	0
COP1	fmt		0		fs		fd		NEG		
010001			00000						000111		
6	5		5		5		5		6		

**Format:** NEG.S fd, fs  
 NEG.D fd, fs

**MIPS32**  
**MIPS32**

**Purpose:**

To negate an FP value

**Description:**  $fd \leftarrow -fs$

The value in FPR *fs* is negated and placed into FPR *fd*. The value is negated by changing the sign bit value. The operand and result are values in format *fmt*. This operation is arithmetic; a NaN operand signals invalid operation.

**Restrictions:**

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; 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, Negate(ValueFPR(fs, fmt)))

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation



---

**No Operation****NOP**

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL	0		0		0		0		SLL		
000000	00000		00000		00000		00000		000000		
6	5		5		5		5		6		

**Format:** NOP**Assembly Idiom****Purpose:**

To perform no operation.

**Description:**

NOP is the assembly idiom used to denote no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 0.

**Restrictions:**

None

**Operation:**

None

**Exceptions:**

None

**Programming Notes:**

The zero instruction word, which represents SLL, r0, r0, 0, is the preferred NOP for software to use to fill branch and jump delay slots and to pad out alignment sequences.

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000						rs			rt		
						rd			0 00000		
									NOR 100111		
6						5			5		

**Format:** NOR *rd*, *rs*, *rt*

**MIPS32**

**Purpose:**

To do a bitwise logical NOT OR

**Description:**  $rd \leftarrow rs \text{ NOR } rt$

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

**Restrictions:**

None

**Operation:**

$GPR[rd] \leftarrow GPR[rs] \text{ nor } GPR[rt]$

**Exceptions:**

None

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000						rs			rt		
						rd			0 00000		
									OR 100101		
6						5			5		

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

**MIPS32**

**Purpose:**

To do a bitwise logical OR

**Description:**  $rd \leftarrow rs \text{ or } rt$

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

**Restrictions:**

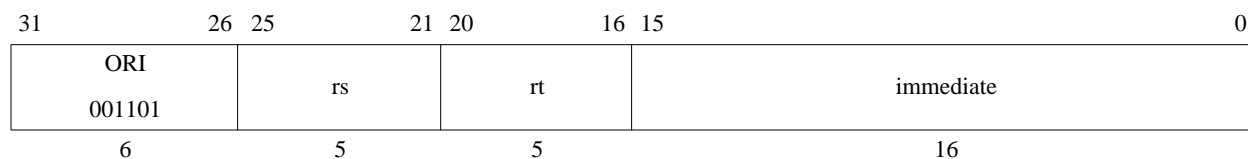
None

**Operation:**

$GPR[rd] \leftarrow GPR[rs] \text{ or } GPR[rt]$

**Exceptions:**

None



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

**MIPS32**

**Purpose:**

To do a bitwise logical OR with a constant

**Description:**  $rt \leftarrow rs \text{ or } \text{immediate}$

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:**

$GPR[rt] \leftarrow GPR[rs] \text{ or } \text{zero\_extend}(\text{immediate})$

**Exceptions:**

None

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

**Format:** `PREF hint,offset(base)`

**MIPS32**

**Purpose:**

To move data between memory and cache.

**Description:** `prefetch_memory(base+offset)`

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

PREF enables the processor to take some action, typically prefetching the data into cache, to improve program performance. The action taken for a specific PREF instruction is both system and context dependent. Any action, including doing nothing, is permitted as long as it does not change architecturally visible state or alter the meaning of a program. Implementations are expected either to do nothing, or to take an action that increases the performance of the program. The PrepareForStore function is unique in that it may modify the architecturally visible state.

PREF does not cause addressing-related exceptions. If the address specified would cause an addressing exception, the exception condition is ignored and no data movement occurs. However even if no data is prefetched, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

PREF never generates a memory operation for a location with an *uncached* memory access type.

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

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

The *hint* field supplies information about the way the data is expected to be used. With the exception of PrepareForStore, a *hint* value cannot cause an action to modify architecturally visible state. A processor may use a *hint* value to improve the effectiveness of the prefetch action.

Table 3-29 Values of the *hint* Field for the PREF Instruction

Value	Name	Data Use and Desired Prefetch Action
0	load	Use: Prefetched data is expected to be read (not modified). Action: Fetch data as if for a load.
1	store	Use: Prefetched data is expected to be stored or modified. Action: Fetch data as if for a store.
2-3	Reserved	Reserved for future use - not available to implementations.
4	load_streamed	Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache. Action: Fetch data as if for a load and place it in the cache so that it does not displace data prefetched as “retained.”
5	store_streamed	Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache. Action: Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as “retained.”
6	load_retained	Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as “streamed.”
7	store_retained	Use: Prefetched data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a store and place it in the cache so that it is not displaced by data prefetched as “streamed.”

**Table 3-29 Values of the *hint* Field for the PREF Instruction**

8-24	Reserved	Reserved for future use - not available to implementations.
25	writeback_invalidate (also known as “nudge”)	<p>Use: Data is no longer expected to be used.</p> <p>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.</p>
26-29	Implementation Dependent	Unassigned by the Architecture - available for implementation-dependent use.
30	PrepareForStore	<p>Use: Prepare the cache for writing an entire line, without the overhead involved in filling the line from memory.</p> <p>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.</p>
31	Implementation Dependent	Unassigned by the Architecture - available for implementation-dependent use.

**Restrictions:**

None

**Operation:**

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

**Exceptions:**

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

**Programming Notes:**

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

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

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.



31	26	25	21	20	16	15	11	10	6	5	0
COP1	fmt		0		fs		fd		ROUND.W		
010001			00000						001100		
6	5		5		5		5		6		

**Format:** ROUND.W.S fd, fs  
 ROUND.W.D fd, fs

**MIPS32**  
**MIPS32**

#### Purpose:

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

**Description:**  $fd \leftarrow \text{convert\_and\_round}(fs)$

The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format rounding to nearest/even (rounding mode 0). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range  $-2^{31}$  to  $2^{31}-1$ , the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result,  $2^{31}-1$ , is written to *fd*.

#### Restrictions:

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

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

#### Operation:

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

**Exceptions:**

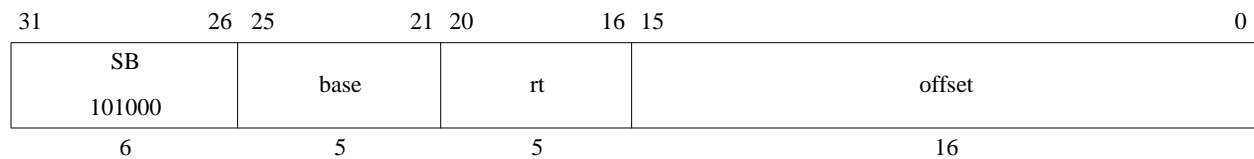
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Unimplemented Operation, Invalid Operation, Overflow

## Store Byte

SB



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

**MIPS32**

### Purpose:

To store a byte to memory

**Description:**  $\text{memory}[\text{base} + \text{offset}] \leftarrow \text{rt}$

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:

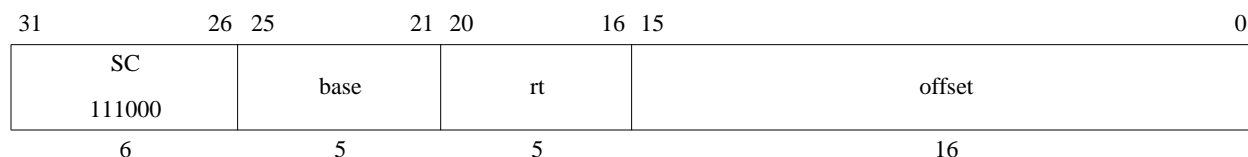
```

vAddr      ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr      ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian2)
bytesel    ← vAddr_1..0 xor BigEndianCPU2
dataword   ← GPR[rt]_31-8*bytesel..0 || 08*bytesel
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)

```

### Exceptions:

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



**Format:** SC *rt*, *offset*(*base*)

**MIPS32**

**Purpose:**

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

**Description:** if *atomic\_update* then *memory*[*base*+*offset*]  $\leftarrow$  *rt*, *rt*  $\leftarrow$  1 else *rt*  $\leftarrow$  0

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

The 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 32-bit word of GPR *rt* is stored into memory at the location specified by the aligned effective address.
- A 1, indicating success, is written into GPR *rt*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *rt*.

If either of the following events occurs between the execution of LL and SC, the SC fails:

- A coherent store is completed by another processor or coherent I/O module into the block of synchronizable physical memory containing the word. The size and alignment of the block is implementation dependent, but it is at least one word and at most the minimum page size.
- An ERET instruction is executed.

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

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

The following conditions must be true or the result of the SC is **UNPREDICTABLE**:

- Execution of SC must have been preceded by execution of an LL instruction.
- An RMW sequence executed without intervening events that would cause the SC to fail must use the same address in the LL and SC. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.

Atomic RMW is provided only for synchronizable memory locations. A synchronizable memory location is one that is associated with the state and logic necessary to implement the LL/SC semantics. Whether a memory location is synchronizable depends on the processor and system configurations, and on the memory access type used for the location:

- **Uniprocessor atomicity:** To provide atomic RMW on a single processor, all accesses to the location must be made with memory access type of either *cached noncoherent* or *cached coherent*. All accesses must be to one or the other access type, and they may not be mixed.
- **MP atomicity:** To provide atomic RMW among multiple processors, all accesses to the location must be made with a memory access type of *cached coherent*.
- **I/O System:** To provide atomic RMW with a coherent I/O system, all accesses to the location must be made with a memory access type of *cached coherent*. If the I/O system does not use coherent memory operations, then atomic RMW cannot be provided with respect to the I/O reads and writes.

#### Restrictions:

The addressed location must have a memory access type of *cached noncoherent* or *cached coherent*; if it does not, the result is **UNPREDICTABLE**.

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

#### Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
if LLbit then
    StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 031 || LLbit

```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

**Programming Notes:**

LL and SC are used to atomically update memory locations, as shown below.

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

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

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

31	26	25	6	5	0
SPECIAL2	code				SDBBP
011100					111111
6	20				6

**Format:** SDBBP code

**EJTAG**

**Purpose:**

To cause a debug breakpoint exception

**Description:**

This instruction causes a debug exception, passing control to the debug exception handler. The code field can be used for passing information to the debug exception handler, and is retrieved by the debug exception handler only by loading the contents of the memory word containing the instruction, using the DEPC register. The CODE field is not used in any way by the hardware.

**Restrictions:**

**Operation:**

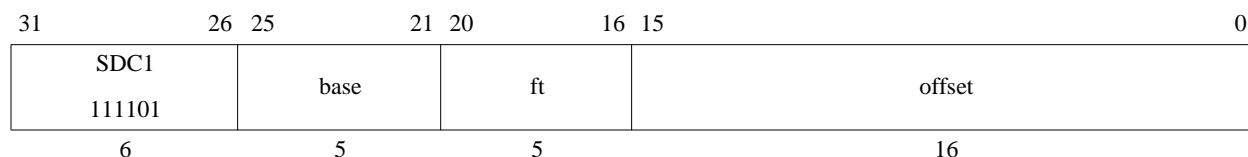
```

If DebugDM = 0 then
    SignalDebugBreakpointException()
else
    SignalDebugModeBreakpointException()
endif

```

**Exceptions:**

Debug Breakpoint Exception



**Format:** SDC1 ft, offset(base)

**MIPS32**

**Purpose:**

To store a doubleword from an FPR to memory

**Description:**  $\text{memory}[\text{base} + \text{offset}] \leftarrow \text{ft}$

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

**Restrictions:**

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

**Operation:**

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
lsw ← ValueFPR(ft, UNINTERPRETED_WORD)
msw ← ValueFPR(ft+1, UNINTERPRETED_WORD)
paddr ← paddr xor ((BigEndianCPU xor ReverseEndian) || 02)

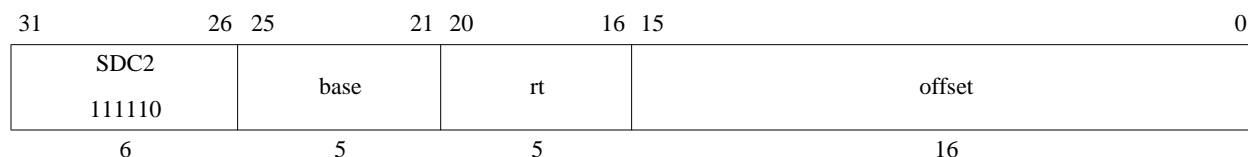
paddr ← paddr xor 2#100

```

**Exceptions:**

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





**Format:** SDC2 rt, offset(base)

**MIPS32**

**Purpose:**

To store a doubleword from a Coprocessor 2 register to memory

**Description:**  $\text{memory}[\text{base} + \text{offset}] \leftarrow \text{rt}$

The 64-bit doubleword in Coprocessor 2 register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

**Restrictions:**

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

**Operation:**

```

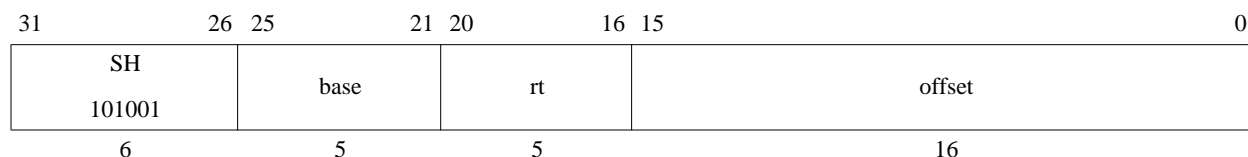
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
lsw ← CPR[2,rt,0]
msw ← CPR[2,rt+1,0]
paddr ← paddr xor ((BigEndianCPU xor ReverseEndian) || 02)

paddr ← paddr xor 2#100

```

**Exceptions:**

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



**Format:** SH *rt*, *offset*(*base*)

**MIPS32**

**Purpose:**

To store a halfword to memory

**Description:**  $\text{memory}[\text{base} + \text{offset}] \leftarrow \text{rt}$

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

**Restrictions:**

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

**Operation:**

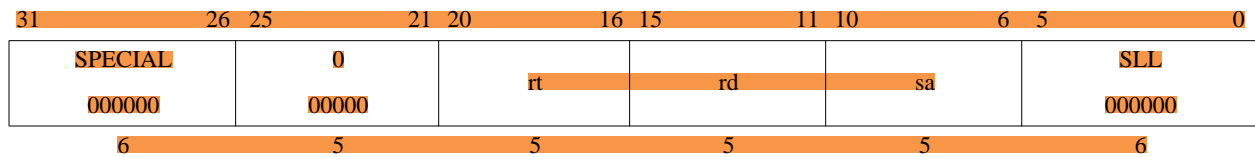
```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor (ReverseEndian || 0))
bytesel ← vAddr1..0 xor (BigEndianCPU || 0)
dataword ← GPR[rt]31-8*bytesel..0 || 08*bytesel
StoreMemory (CCA, HALFWORD, dataword, pAddr, vAddr, DATA)

```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

**Format:** SLL rd, rt, sa**MIPS32****Purpose:**

To left-shift a word by a fixed number of bits

**Description:**  $rd \leftarrow rt \ll sa$ 

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 placed in GPR *rd*. The bit-shift amount is specified by *sa*.

**Restrictions:**

None

**Operation:**

```

s      ← sa
temp   ← GPR[rt]_(31-s)..0 || 0s
GPR[rd] ← temp

```

**Exceptions:**

None

**Programming Notes:**

SLL r0, r0, 0, expressed as NOP, is the assembly idiom used to denote no operation.

SLL r0, r0, 1, expressed as SSNOP, is the assembly idiom used to denote no operation that causes an issue break on superscalar processors.

## Shift Word Left Logical Variable

SLLV

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL						rs					
000000						rt					
						rd					
						0					
						SLLV					
						00000					
						000100					
6						5					

**Format:** SLLV rd, rt, rs**MIPS32****Purpose:** To left-shift a word by a variable number of bits**Description:**  $rd \leftarrow rt \ll rs$ 

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 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] ← temp

```

**Exceptions:** None**Programming Notes:**

None

## Set on Less Than

SLT

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000	rs		rt		rd		0 00000		SLT 101010		
6	5		5		5		5		6		

**Format:** SLT rd, rs, rt

**MIPS32**

### Purpose:

To record the result of a less-than comparison

**Description:**  $rd \leftarrow (rs < rt)$

Compare the contents of GPR *rs* and GPR *rt* as signed integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

### Restrictions:

None

### Operation:

```

if GPR[rs] < GPR[rt] then
    GPR[rd] ← 0GPRLEN-1 || 1
else
    GPR[rd] ← 0GPRLEN
endif

```

### Exceptions:

None

31	26	25	21	20	16	15	0
SLTI 001010		rs		rt		immediate	
6		5		5		16	

**Format:** SLTI *rt*, *rs*, *immediate*

**MIPS32**

**Purpose:**

To record the result of a less-than comparison with a constant

**Description:**  $rt \leftarrow (rs < immediate)$

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**

None

**Operation:**

```

if GPR[rs] < sign_extend(immediate) then
    GPR[rd] ← 0GPREN-1 || 1
else
    GPR[rd] ← 0GPREN
endif

```

**Exceptions:**

None

31	26	25	21	20	16	15	0
SLTIU		rs		rt		immediate	
001011							
6		5		5		16	

**Format:** SLTIU *rt*, *rs*, *immediate*

**MIPS32**

**Purpose:**

To record the result of an unsigned less-than comparison with a constant

**Description:**  $rt \leftarrow (rs < immediate)$

Compare the contents of GPR *rs* and the sign-extended 16-bit *immediate* as unsigned integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max\_unsigned-32767, max\_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**

None

**Operation:**

```

if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
    GPR[rd] ← 0GPREN-1 || 1
else
    GPR[rd] ← 0GPREN
endif

```

**Exceptions:**

None

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000						rs			rt		
						rd			0 00000		
6						5			5		
									SLTU 101011		
									6		

**Format:** SLTU rd, rs, rt

**MIPS32**

**Purpose:**

To record the result of an unsigned less-than comparison

**Description:**  $rd \leftarrow (rs < rt)$

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**

None

**Operation:**

```

if (0 || GPR[rs]) < (0 || GPR[rt]) then
    GPR[rd] ← 0GPREN-1 || 1
else
    GPR[rd] ← 0GPREN
endif

```

**Exceptions:**

None



31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	0 00000	fs	fd	SQRT 000100	
6	5	5	5	5	6	

**Format:** SQRT.S fd, fs  
SQRT.D fd, fs

**MIPS32**  
**MIPS32**

**Purpose:**

To compute the square root of an FP value

**Description:**  $fd \leftarrow \text{SQRT}(fs)$

The square root of the value in FPR *fs* is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operand and result are values in format *fmt*.

If the value in FPR *fs* corresponds to  $-0$ , the result is  $-0$ .

**Restrictions:**

If the value in FPR *fs* is less than 0, an Invalid Operation condition is raised.

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

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

**Operation:**

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

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Inexact, Unimplemented Operation

## Shift Word Right Arithmetic

**SRA**

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL	0		rt		rd		sa		SRA		
000000	00000								000011		
6	5		5		5		5		6		

**Format:** SRA rd, rt, sa

**MIPS32**

### Purpose:

To execute an arithmetic right-shift of a word by a fixed number of bits

**Description:**  $rd \leftarrow rt \gg sa$  (arithmetic)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

### Restrictions:

None

### Operation:

```

s      ← sa
temp   ← (GPR[rt]31)s || GPR[rt]31..s
GPR[rd] ← temp

```

**Exceptions:** None

## Shift Word Right Arithmetic Variable

SRAV

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL						rs					
000000						rt					
						rd					
						0					
						SRAV					
						00000					
						000111					
6						5					

**Format:** SRAV rd, rt, rs

**MIPS32**

### Purpose:

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

**Description:**  $rd \leftarrow rt \gg rs$  (arithmetic)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift 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 || GPR[rt]31..s
GPR[rd] ← temp

```

### Exceptions:

None

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000			0 00000		rt		rd		sa		SRL 000010
6			5		5		5		5		6

**Format:** SRL *rd*, *rt*, *sa*

**MIPS32**

**Purpose:**

To execute a logical right-shift of a word by a fixed number of bits

**Description:**  $rd \leftarrow rt \gg sa$  (logical)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

**Restrictions:**

None

**Operation:**

```

s      ← sa
temp   ← 0s || GPR[rt]31..s
GPR[rd] ← temp

```

**Exceptions:**

None

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL						rs					
000000						rt					
						rd					
						0					
						00000					
						SRLV					
						000110					
6						5					
						5					
						5					
						5					
						5					
						6					

**Format:** SRLV rd, rt, rs

**MIPS32**

**Purpose:**

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

**Description:**  $rd \leftarrow rt \gg rs$  (logical)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

**Restrictions:**

None

**Operation:**

```

s      ← GPR[rs]4..0
temp   ← 0s || GPR[rt]31..s
GPR[rd] ← temp

```

**Exceptions:**

None

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL	0		0		0		1		SLL		
000000	00000		00000		00000		00001		000000		
6	5		5		5		5		6		

**Format:** SSNOP**MIPS32****Purpose:**

Break superscalar issue on a superscalar processor.

**Description:**

SSNOP is the assembly idiom used to denote superscalar no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 1.

This instruction alters the instruction issue behavior on a superscalar processor by forcing the SSNOP instruction to single-issue. The processor must then end the current instruction issue between the instruction previous to the SSNOP and the SSNOP. The SSNOP then issues alone in the next issue slot.

On a single-issue processor, this instruction is a NOP that takes an issue slot.

**Restrictions:**

None

**Operation:**

None

**Exceptions:**

None

**Programming Notes:**

SSNOP is intended for use primarily to allow the programmer control over CP0 hazards by converting instructions into cycles in a superscalar processor. For example, to insert at least two cycles between an MTC0 and an ERET, one would use the following sequence:

```
mtc0    x,y
ssnop
ssnop
eret
```

Based on the normal issues rules of the processor, the MTC0 issues in cycle T. Because the SSNOP instructions must issue alone, they may issue no earlier than cycle T+1 and cycle T+2, respectively. Finally, the ERET issues no earlier than cycle T+3. Note that although the instruction after an SSNOP may issue no earlier than the cycle after the SSNOP is issued, that instruction may issue later. This is because other implementation-dependent issue rules may apply that prevent an issue in the next cycle. Processors should not introduce any unnecessary delay in issuing SSNOP instructions.

**Subtract Word****SUB**

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

**Format:** SUB *rd*, *rs*, *rt***MIPS32****Purpose:**

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

**Description:**  $rd \leftarrow rs - rt$ 

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

**Restrictions:****None****Operation:**

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

**Exceptions:**

Integer Overflow

**Programming Notes:**

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

31	26	25	21	20	16	15	11	10	6	5	0
COP1 010001			fmt		ft		fs		fd		SUB 000001
6			5		5		5		5		6

**Format:** SUB.S fd, fs, ft  
SUB.D fd, fs, ft

**MIPS32**  
**MIPS32**

**Purpose:**

To subtract FP values

**Description:**  $fd \leftarrow fs - ft$

The value in FPR *ft* is subtracted from the value in FPR *fs*. The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. **Restrictions:**

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

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

**Operation:**

StoreFPR (fd, fmt, ValueFPR(fs, fmt) -<sub>fmt</sub> ValueFPR(ft, fmt))

**CPU Exceptions:**

Coprocessor Unusable, Reserved Instruction

**FPU Exceptions:**

Inexact, Overflow, Underflow, Invalid Op, Unimplemented Op



## Subtract Unsigned Word

SUBU

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL	rs		rt		rd		0		SUBU		
000000							00000		100011		
6	5		5		5		5		6		

**Format:** SUBU rd, rs, rt

**MIPS32**

### Purpose:

To subtract 32-bit integers

**Description:**  $rd \leftarrow rs - rt$

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

No integer overflow exception occurs under any circumstances.

### Restrictions:

None

### Operation:

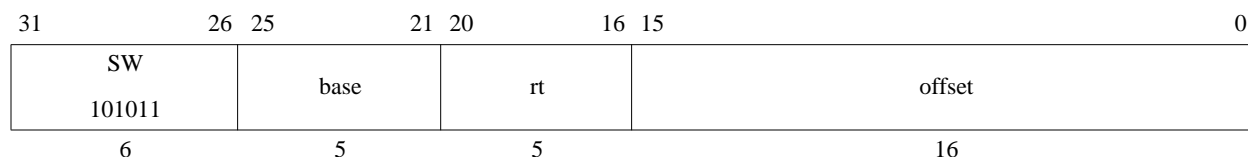
```
temp ← GPR[rs] - GPR[rt]
GPR[rd] ← temp
```

### Exceptions:

None

### Programming Notes:

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



**Format:** SW *rt*, *offset*(*base*)

**MIPS32**

**Purpose:**

To store a word to memory

**Description:**  $\text{memory}[\text{base} + \text{offset}] \leftarrow \text{rt}$

The least-significant 32-bit word of register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

**Restrictions:**

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

**Operation:**

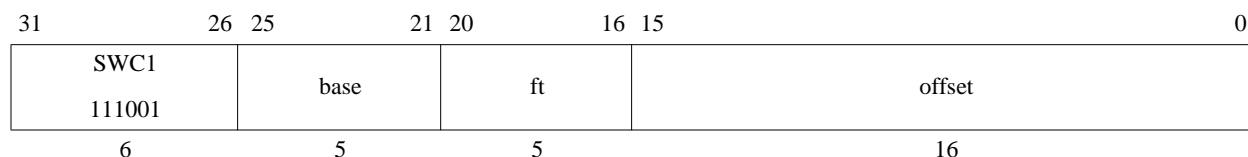
```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)

```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch



**Format:** SWC1 ft, offset(base)

**MIPS32**

**Purpose:**

To store a word from an FPR to memory

**Description:**  $\text{memory}[\text{base} + \text{offset}] \leftarrow \text{ft}$

The low 32-bit word from FPR *ft* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

**Restrictions:**

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

**Operation:**

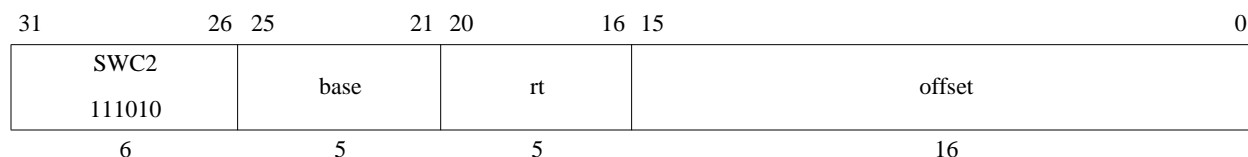
```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
dataword ← ValueFPR(ft, UNINTERPRETED_WORD)
StoreMemory(CCA, WORD, dataword, pAddr, vAddr, DATA)

```

**Exceptions:**

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



**Format:** SWC2 *rt*, *offset*(*base*)

**MIPS32**

**Purpose:**

To store a word from a COP2 register to memory

**Description:**  $\text{memory}[\text{base} + \text{offset}] \leftarrow \text{rt}$

The low 32-bit word from COP2 (Coprocessor 2) register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

**Restrictions:**

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

**Operation:**

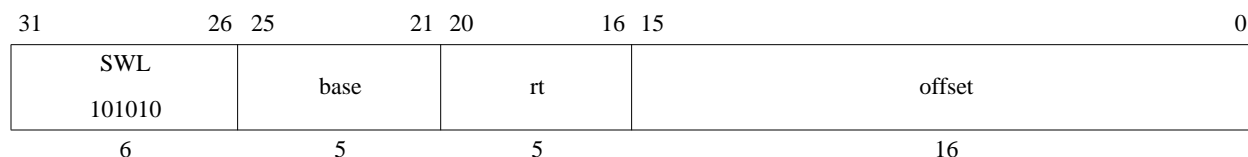
```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
dataword ← CPR[2,rt,0]
StoreMemory(CCA, WORD, dataword, pAddr, vAddr, DATA)

```

**Exceptions:**

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



**Format:** SWL rt, offset(base)

**MIPS32**

**Purpose:**

To store the most-significant part of a word to an unaligned memory address

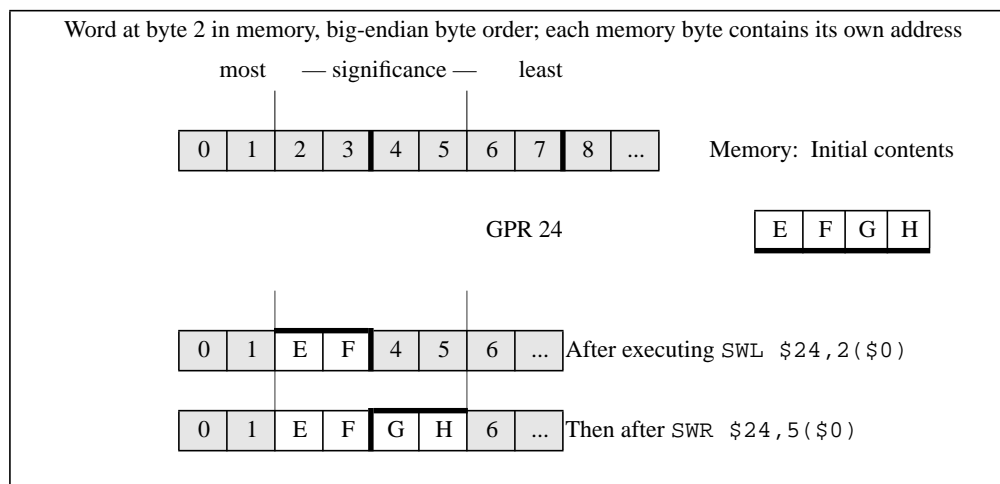
**Description:**  $\text{memory}[\text{base} + \text{offset}] \leftarrow \text{rt}$

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*.

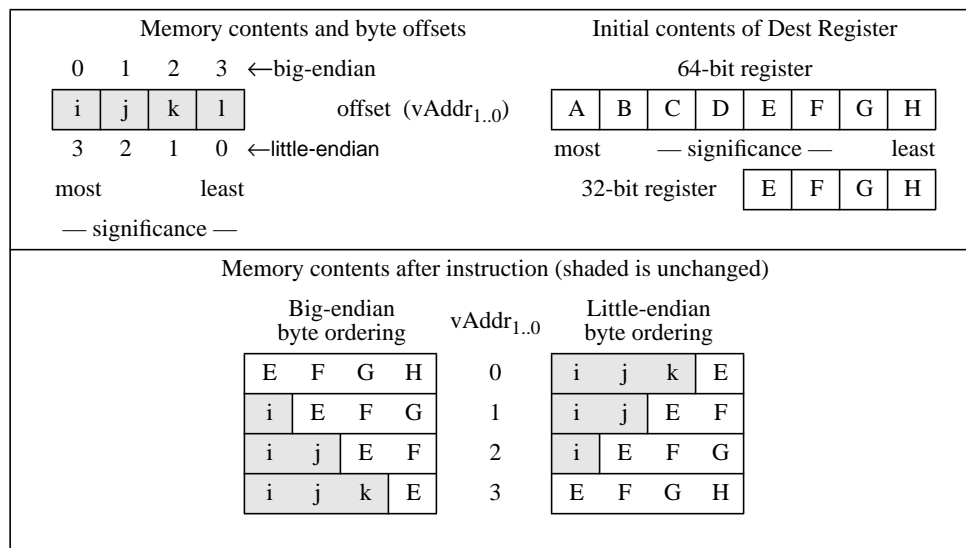
The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWL stores the most-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWR stores the remainder of the unaligned word.

**Figure 3-6 Unaligned Word Store Using SWL and SWR**



The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address (*vAddr1..0*)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte ordering.

Figure 3-7 Bytes Stored by an SWL Instruction

**Restrictions:**

None

**Operation:**

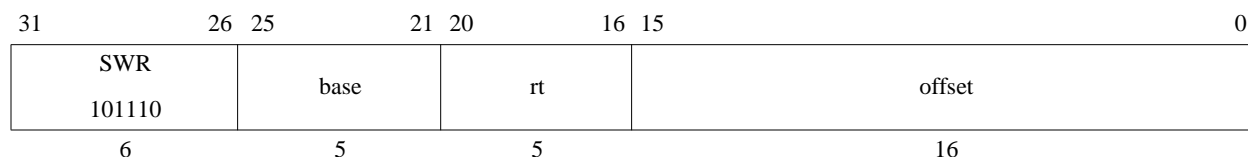
```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian2)
If BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..2 || 02
endif
byte ← vAddr1..0 xor BigEndianCPU2
dataword ← 024-8*byte || GPR[rt]31..24-8*byte
StoreMemory(CCA, byte, dataword, pAddr, vAddr, DATA)

```

**Exceptions:**

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



**Format:** SWR *rt*, *offset*(*base*)

**MIPS32**

**Purpose:**

To store the least-significant part of a word to an unaligned memory address

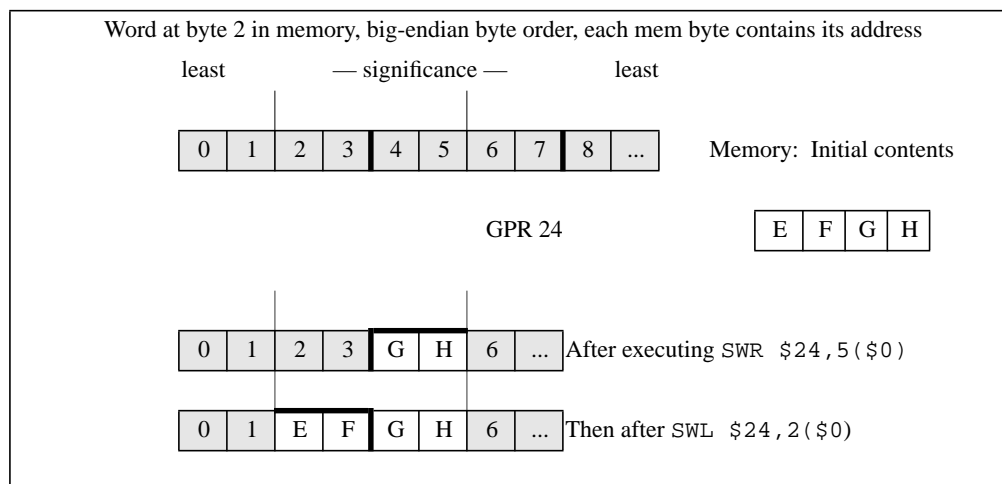
**Description:**  $\text{memory}[\text{base} + \text{offset}] \leftarrow \text{rt}$

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*.

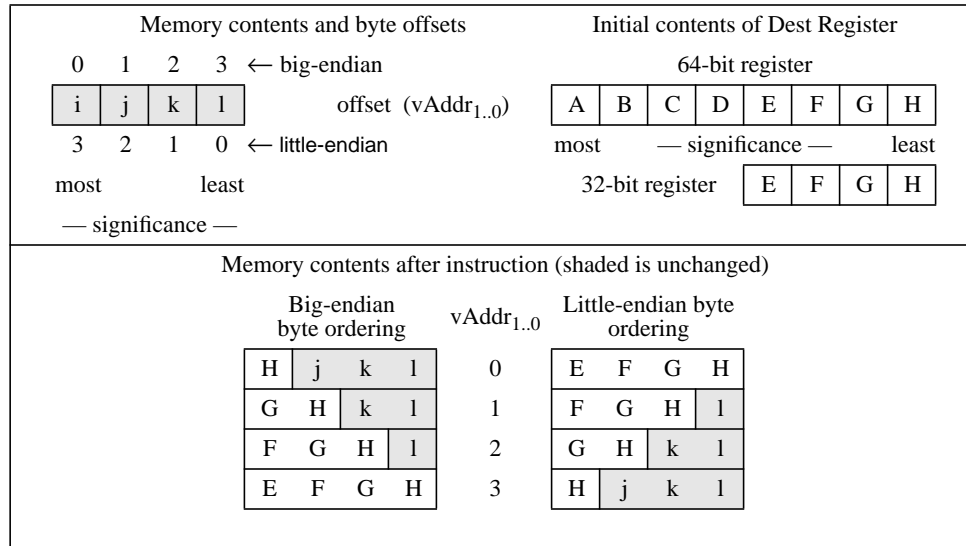
The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR stores the least-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWL stores the remainder of the unaligned word.

**Figure 3-8 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 3-9 Bytes Stored by SWR Instruction**



#### Restrictions:

None

#### Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian2)
If BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..2 || 02
endif
byte ← vAddr1..0 xor BigEndianCPU2
dataword ← GPR[rt]31-8*byte || 08*byte
StoreMemory(CCA, WORD-byte, dataword, pAddr, vAddr, DATA)

```

#### Exceptions:

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



31	26	25	21	20	16	15	11	10	6	5	0	
SPECIAL			0						stype		SYNC	
000000			00 0000 0000 0000 0								001111	
6			15						5		6	

**Format:** SYNC (stype = 0 implied)

**MIPS32**

**Purpose:**

To order loads and stores.

**Description:**

*Simple Description:*

- SYNC affects only *uncached* and *cached coherent* loads and stores. The loads and stores that occur before the SYNC must be completed before the loads and stores after the SYNC are allowed to start.
- Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.
- SYNC is required, potentially in conjunction with SSNOP, to guarantee that memory reference results are visible across operating mode changes. For example, a SYNC is required on some implementations on entry to and exit from Debug Mode to guarantee that memory affects are handled correctly.

*Detailed Description:*

- When the *stype* field has a value of zero, every synchronizable load and store that occurs in the instruction stream before the SYNC instruction must be globally performed before any synchronizable load or store that occurs after the SYNC can be performed, with respect to any other processor or coherent I/O module.
- SYNC does not guarantee the order in which instruction fetches are performed. The *stype* values 1-31 are reserved for future extensions to the architecture. A value of zero will always be defined such that it performs all defined synchronization operations. Non-zero values may be defined to remove some synchronization operations. As such, software should never use a non-zero value of the *stype* field, as this may inadvertently cause future failures if non-zero values remove synchronization operations.

**Terms:**

*Synchronizable:* A load or store instruction is *synchronizable* if the load or store occurs to a physical location in shared memory using a virtual location with a memory access type of either *uncached* or *cached coherent*. *Shared memory* is memory that can be accessed by more than one processor or by a coherent I/O system module.

*Performed load:* A load instruction is *performed* when the value returned by the load has been determined. The result of a load on processor A has been *determined* with respect to processor or coherent I/O module B when a subsequent store to the location by B cannot affect the value returned by the load. The store by B must use the same memory access type as the load.

*Performed store:* A store instruction is *performed* when the store is observable. A store on processor A is *observable* with respect to processor or coherent I/O module B when a subsequent load of the location by B returns the value written by the store. The load by B must use the same memory access type as the store.

*Globally performed load:* A load instruction is *globally performed* when it is performed with respect to all processors and coherent I/O modules capable of storing to the location.

*Globally performed store:* A store instruction is *globally performed* when it is globally observable. It is *globally observable* when it is observable by all processors and I/O modules capable of loading from the location.

*Coherent I/O module:* A *coherent I/O module* is an Input/Output system component that performs coherent Direct Memory Access (DMA). It reads and writes memory independently as though it were a processor doing loads and stores to locations with a memory access type of *cached coherent*.

**Restrictions:**

The effect of SYNC on the global order of loads and stores for memory access types other than *uncached* and *cached coherent* is **UNPREDICTABLE**.

**Operation:**

`SyncOperation(stype)`

**Exceptions:**

None

**Programming Notes:**

A processor executing load and store instructions observes the order in which loads and stores using the same memory access type occur in the instruction stream; this is known as *program order*.

A *parallel program* has multiple instruction streams that can execute simultaneously on different processors. In multiprocessor (MP) systems, the order in which the effects of loads and stores are observed by other processors—the *global order* of the loads and store—determines the actions necessary to reliably share data in parallel programs.

When all processors observe the effects of loads and stores in program order, the system is *strongly ordered*. On such systems, parallel programs can reliably share data without explicit actions in the programs. For such a system, SYNC has the same effect as a NOP. Executing SYNC on such a system is not necessary, but neither is it an error.

If a multiprocessor system is not strongly ordered, the effects of load and store instructions executed by one processor may be observed out of program order by other processors. On such systems, parallel programs must take explicit actions to reliably share data. At critical points in the program, the effects of loads and stores from an instruction stream must occur in the same order for all processors. SYNC separates the loads and stores executed on the processor into two groups, and the effect of all loads and stores in one group is seen by all processors before the effect of any load or store in the subsequent group. In effect, SYNC causes the system to be strongly ordered for the executing processor at the instant that the SYNC is executed.

Many MIPS-based multiprocessor systems are strongly ordered or have a mode in which they operate as strongly ordered for at least one memory access type. The MIPS architecture also permits implementation of MP systems that are not strongly ordered; SYNC enables the reliable use of shared memory on such systems. A parallel program that does not use SYNC generally does not operate on a system that is not strongly ordered. However, a program that does use SYNC works on both types of systems. (System-specific documentation describes the actions needed to reliably share data in parallel programs for that system.)

The behavior of a load or store using one memory access type is **UNPREDICTABLE** if a load or store was previously made to the same physical location using a different memory access type. The presence of a SYNC between the references does not alter this behavior.

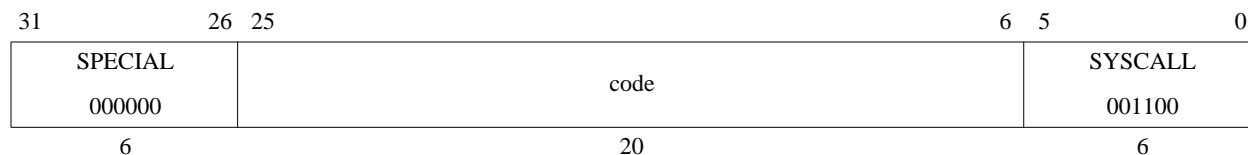
SYNC affects the order in which the effects of load and store instructions appear to all processors; it does not generally affect the physical memory-system ordering or synchronization issues that arise in system programming. The effect of SYNC on implementation-specific aspects of the cached memory system, such as writeback buffers, is not defined. The effect of SYNC on reads or writes to memory caused by privileged implementation-specific instructions, such as CACHE, also is not defined.

```
# Processor A (writer)
# Conditions at entry:
# The value 0 has been stored in FLAG and that value is observable by B
SW    R1, DATA      # change shared DATA value
LI    R2, 1
SYNC                      # Perform DATA store before performing FLAG store
SW    R2, FLAG        # say that the shared DATA value is valid

# Processor B (reader)
LI    R2, 1
1: LW   R1, FLAG      # Get FLAG
BNE    R2, R1, 1B     # if it says that DATA is not valid, poll again
NOP
SYNC                      # FLAG value checked before doing DATA read
LW    R1, DATA      # Read (valid) shared DATA value
```

Prefetch operations have no effect detectable by User-mode programs, so ordering the effects of prefetch operations is not meaningful.

The code fragments above shows how SYNC can be used to coordinate the use of shared data between separate writer and reader instruction streams in a multiprocessor environment. The FLAG location is used by the instruction streams to determine whether the shared data item DATA is valid. The SYNC executed by processor A forces the store of DATA to be performed globally before the store to FLAG is performed. The SYNC executed by processor B ensures that DATA is not read until after the FLAG value indicates that the shared data is valid.

**Format:** SYSCALL**MIPS32****Purpose:**

To cause a System Call exception

**Description:**

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

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

**Restrictions:**

None

**Operation:**

```
SignalException(SystemCall)
```

**Exceptions:**

System Call

---

**Trap if Equal****TEQ**

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000			rs		rt		code		
6			5		5		10		
							TEQ 110100		
							6		

**Format:** TEQ *rs*, *rt***MIPS32****Purpose:**

To compare GPRs and do a conditional trap

**Description:** if *rs* = *rt* then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**

None

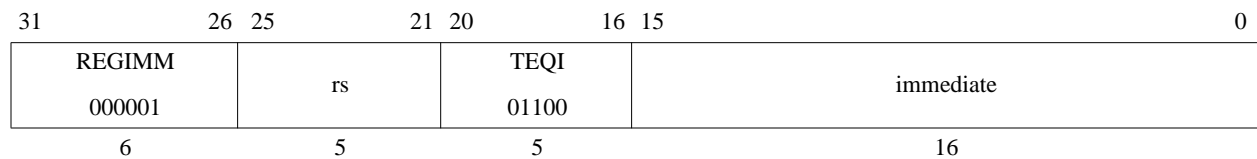
**Operation:**

```
if GPR[rs] = GPR[rt] then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

---

**Trap if Equal Immediate****TEQI****Format:** TEQI *rs*, *immediate***MIPS32****Purpose:**

To compare a GPR to a constant and do a conditional trap

**Description:** if *rs* = *immediate* then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is equal to *immediate*, then take a Trap exception.

**Restrictions:**

None

**Operation:**

```
if GPR[rs] = sign_extend(immediate) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

## Trap if Greater or Equal

TGE

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000	rs		rt		code				TGE 110000
6	5		5		10				6

**Format:** TGE *rs*, *rt*

**MIPS32**

### Purpose:

To compare GPRs and do a conditional trap

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

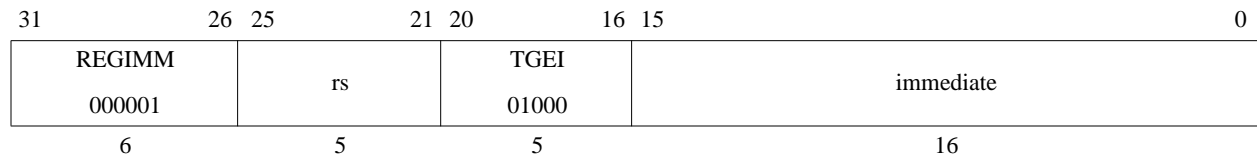
```
if GPR[rs]  $\geq$  GPR[rt] then
    SignalException(Trap)
endif
```

### Exceptions:

Trap



---

**Trap if Greater or Equal Immediate****TGEI****Format:** TGEI rs, immediate**MIPS32****Purpose:**

To compare a GPR to a constant and do a conditional trap

**Description:** if  $rs \geq \text{immediate}$  then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is greater than or equal to *immediate*, then take a Trap exception.

**Restrictions:**

None

**Operation:**

```
if GPR[rs] ≥ sign_extend(immediate) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

## Trap if Greater or Equal Immediate Unsigned

TGEIU

31	26	25	21	20	16	15	0
REGIMM			rs		TGEIU		immediate
000001					01001		
6			5		5		16

**Format:** TGEIU rs, immediate

**MIPS32**

### Purpose:

To compare a GPR to a constant and do a conditional trap

**Description:** if  $rs \geq \text{immediate}$  then Trap

Compare the contents of GPR *rs* and the 16-bit sign-extended *immediate* as unsigned integers; if GPR *rs* is greater than or equal to *immediate*, then take a Trap exception.

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max\_unsigned-32767, max\_unsigned] end of the unsigned range.

### Restrictions:

None

### Operation:

```
if (0 || GPR[rs]) ≥ (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

### Exceptions:

Trap

---

**Trap if Greater or Equal Unsigned****TGEU**

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000			rs		rt		code		TGEU 110001
6			5		5		10		6

**Format:** TGEU *rs*, *rt***MIPS32****Purpose:**

To compare GPRs and do a conditional trap

**Description:** if *rs* ≥ *rt* then Trap

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; if GPR *rs* is greater than or equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**

None

**Operation:**

```
if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

## Probe TLB for Matching Entry

TLBP

31	26	25	24		6	5	0
COP0	CO	0				TLBP	
010000	1	000 0000 0000 0000 0000				001000	
6	1	19				6	

**Format:** TLBP

**MIPS32**

### Purpose:

To find a matching entry in the TLB.

### Description:

The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the *Index* register is set.

### Restrictions:

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

### 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]G = 1) or (TLB[i]ASID = EntryHiASID)) then
        Index ← i
    endif
endfor

```

### Exceptions:

Coprocessor Unusable

## Read Indexed TLB Entry

TLBR

31	26	25	24		6	5	0
COP0	CO	0				TLBR	
010000	1	000 0000 0000 0000 0000				000001	
6	1	19				6	

**Format:** TLBR

**MIPS32**

### Purpose:

To read an entry from the TLB.

### Description:

The *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers are loaded with the contents of the TLB entry pointed to by the Index register. Note that the value written to the *EntryHi*, *EntryLo0*, and *EntryLo1* registers may be different from that originally written to the TLB via these registers in that:

- The value returned in the VPN2 field of the *EntryHi* register may have those 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.

**Operation:**

```

i ← Index
if i > (TLBEntries - 1) then
    UNDEFINED
endif
PageMaskMask ← TLB[i]Mask
EntryHi ←
    (TLB[i]VPN2 and not TLB[i]Mask) || # Masking implementation dependent
    05 || TLB[i]ASID
EntryLo1 ← 02 ||
    (TLB[i]PFN1 and not TLB[i]Mask) || # Masking mplementation dependent
    TLB[i]C1 || TLB[i]D1 || TLB[i]V1 || TLB[i]G
EntryLo0 ← 02 ||
    (TLB[i]PFN0 and not TLB[i]Mask) || # Masking mplementation dependent
    TLB[i]C0 || TLB[i]D0 || TLB[i]V0 || TLB[i]G

```

**Exceptions:**

Coprocessor Unusable

**Write Indexed TLB Entry****TLBWI**

31	26	25	24		6	5	0
COP0	CO	0				TLBWI	
010000	1	000 0000 0000 0000 0000				000010	
6	1	19				6	

**Format:** TLBWI**MIPS32****Purpose:**

To write a TLB entry indexed by the *Index* register.

**Description:**

The TLB entry pointed to by the *Index* register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the *PageMask* register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of *PageMask* register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.

**Restrictions:**

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of TLB entries in the processor.

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

**Operation:**

```

i ← Index
TLB[i]Mask ← PageMaskMask
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



**Write Random TLB Entry****TLBWR**

31	26	25	24		6	5	0
COP0		CO	0			TLBWR	
010000		1	000 0000 0000 0000 0000			000110	
6		1	19			6	

**Format:** TLBWR**MIPS32****Purpose:**

To write a TLB entry indexed by the *Random* register.

**Description:**

The TLB entry pointed to by the *Random* register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the *PageMask* register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of *PageMask* register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.

**Restrictions:**

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

**Operation:**

```

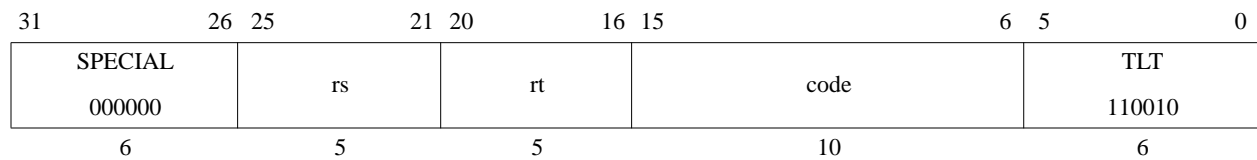
i ← Random
TLB[i]Mask ← PageMaskMask
TLB[i]VPN2 ← EntryHiVPN2 and not PageMaskMask # Implementation dependent
TLB[i]ASID ← EntryHiASID
TLB[i]G ← EntryLo1G and EntryLo0G
TLB[i]PFN1 ← EntryLo1PFN and not PageMaskMask # Implementation dependent
TLB[i]C1 ← EntryLo1C
TLB[i]D1 ← EntryLo1D
TLB[i]V1 ← EntryLo1V
TLB[i]PFN0 ← EntryLo0PFN and not PageMaskMask # Implementation dependent
TLB[i]C0 ← EntryLo0C
TLB[i]D0 ← EntryLo0D
TLB[i]V0 ← EntryLo0V

```

**Exceptions:**

Coprocessor Unusable

---

**Trap if Less Than****TLT****Format:** TLT *rs*, *rt***MIPS32****Purpose:**

To compare GPRs and do a conditional trap

**Description:** if *rs* < *rt* then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is less than GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**

None

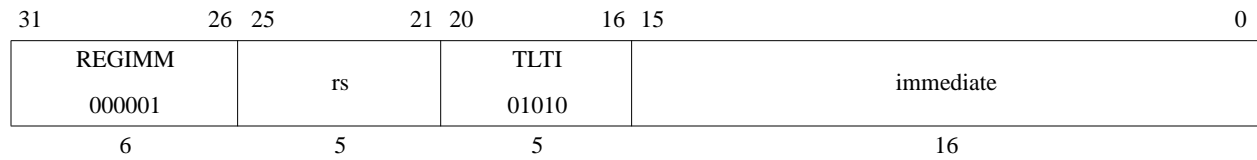
**Operation:**

```
if GPR[rs] < GPR[rt] then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

---

**Trap if Less Than Immediate****TLTI****Format:** TLTI rs, immediate**MIPS32****Purpose:**

To compare a GPR to a constant and do a conditional trap

**Description:** if *rs* < *immediate* then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is less than *immediate*, then take a Trap exception.

**Restrictions:**

None

**Operation:**

```
if GPR[rs] < sign_extend(immediate) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

## Trap if Less Than Immediate Unsigned

TLTIU

31	26	25	21	20	16	15	0
REGIMM 000001			rs		TLTIU 01011		immediate
6			5		5		16

**Format:** TLTIU rs, immediate

**MIPS32**

### Purpose:

To compare a GPR to a constant and do a conditional trap

**Description:** if *rs* < *immediate* then Trap

Compare the contents of GPR *rs* and the 16-bit sign-extended *immediate* as unsigned integers; if GPR *rs* is less than *immediate*, then take a Trap exception.

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max\_unsigned-32767, max\_unsigned] end of the unsigned range.

### Restrictions:

None

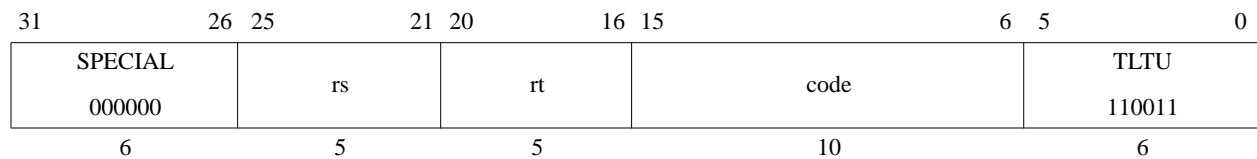
### Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

### Exceptions:

Trap

---

**Trap if Less Than Unsigned****TLTU****Format:** TLTU *rs*, *rt***MIPS32****Purpose:**

To compare GPRs and do a conditional trap

**Description:** if *rs* < *rt* then TrapCompare the contents of GPR *rs* and GPR *rt* as unsigned integers; if GPR *rs* is less than GPR *rt*, then take a Trap exception.The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.**Restrictions:**

None

**Operation:**

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

---

**Trap if Not Equal****TNE**

31	26	25	21	20	16	15	6	5	0
SPECIAL 000000			rs		rt		code		
6			5		5		10		
							TNE 110110		
							6		

**Format:** TNE *rs*, *rt***MIPS32****Purpose:**

To compare GPRs and do a conditional trap

**Description:** if *rs*  $\neq$  *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

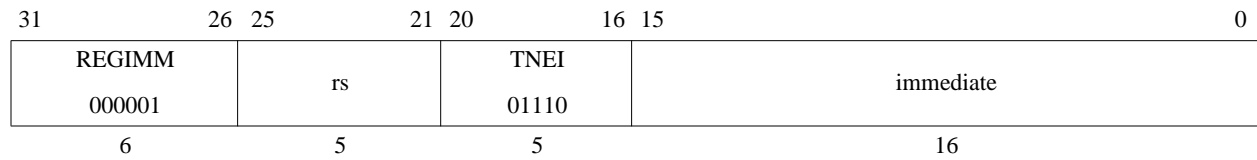
**Operation:**

```
if GPR[rs]  $\neq$  GPR[rt] then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap

---

**Trap if Not Equal****TNEI****Format:** TNEI *rs*, *immediate***MIPS32****Purpose:**

To compare a GPR to a constant and do a conditional trap

**Description:** if *rs*  $\neq$  *immediate* then TrapCompare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is not equal to *immediate*, then take a Trap exception.**Restrictions:**

None

**Operation:**

```
if GPR[rs]  $\neq$  sign_extend(immediate) then
    SignalException(Trap)
endif
```

**Exceptions:**

Trap



31	26	25	21	20	16	15	11	10	6	5	0
COP1	fmt		0		fs		fd		TRUNC.W		
010001			00000						001101		
6	5		5		5		5		6		

**Format:** TRUNC.W.S fd, fs  
 TRUNC.W.D fd, fs

**MIPS32**  
**MIPS32**

### Purpose:

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

**Description:**  $fd \leftarrow \text{convert\_and\_round}(fs)$

The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format using rounding toward zero (rounding mode 1). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range  $-2^{31}$  to  $2^{31}-1$ , the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result,  $2^{31}-1$ , is written to *fd*.

### Restrictions:

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

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

### Operation:

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

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Invalid Operation, Overflow, Unimplemented Operation

---

**Enter Standby Mode****WAIT**

31	26	25	24		6	5	0
COP0		CO	Implementation-Dependent Code			WAIT	
010000		1				100000	
6		1	19			6	

**Format:** WAIT**MIPS32****Purpose:**

Wait for Event

**Description:**

The WAIT instruction performs an implementation-dependent operation, usually involving a lower power mode. Software may use bits 24:6 of the instruction to communicate additional information to the processor, and the processor may use this information as control for the lower power mode. A value of zero for bits 24:6 is the default and must be valid in all implementations.

The WAIT instruction is typically implemented by stalling the pipeline at the completion of the instruction and entering a lower power mode. The pipeline is restarted when an external event, such as an interrupt or external request occurs, and execution continues with the instruction following the WAIT instruction. It is implementation-dependent whether the pipeline restarts when a non-enabled interrupt is requested. In this case, software must poll for the cause of the restart. If the pipeline restarts as the result of an enabled interrupt, that interrupt is taken between the WAIT instruction and the following instruction (EPC for the interrupt points at the instruction following the WAIT instruction).

The assertion of any reset or NMI must restart the pipeline and the corresponding exception must be taken.

**Restrictions:**

The operation of the processor is **UNDEFINED** if a WAIT instruction is placed in the delay slot of a branch or a jump.

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

**Operation:**

Enter implementation dependent lower power mode

**Exceptions:**

Coprocessor Unusable Exception

---

**Exclusive OR****XOR**

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000						rs		rt		rd	
0						00000		XOR		100110	
6						5		5		5	

**Format:** XOR *rd*, *rs*, *rt***MIPS32****Purpose:**

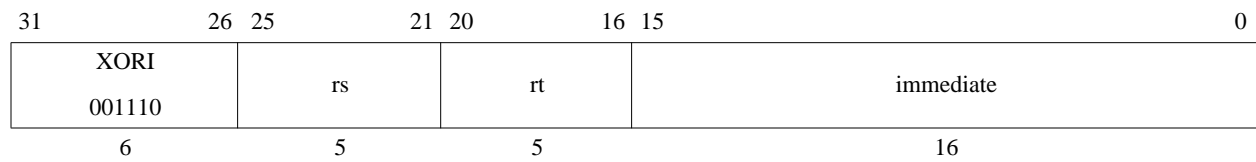
To do a bitwise logical Exclusive OR

**Description:**  $rd \leftarrow rs \text{ XOR } rt$ Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical Exclusive OR operation and place the result into GPR *rd*.**Restrictions:**

None

**Operation:**
$$GPR[rd] \leftarrow GPR[rs] \text{ xor } GPR[rt]$$
**Exceptions:**

None



**Format:** XORI *rt*, *rs*, *immediate*

**MIPS32**

**Purpose:**

To do a bitwise logical Exclusive OR with a constant

**Description:**  $rt \leftarrow rs \text{ XOR } immediate$

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:**

$GPR[rt] \leftarrow GPR[rs] \text{ xor } zero\_extend(immediate)$

**Exceptions:**

None

## Instruction Bit Encodings

### A.1 Instruction Encodings and Instruction Classes

Instruction encodings are presented in this section; field names are printed here and throughout the book in *italics*.

When encoding an instruction, the primary *opcode* field is encoded first. Most *opcode* values completely specify an instruction that has an *immediate* value or offset.

*Opcode* values that do not specify an instruction instead specify an instruction class. Instructions within a class are further specified by values in other fields. For instance, *opcode* REGIMM specifies the *immediate* instruction class, which includes conditional branch and trap *immediate* instructions.

### A.2 Instruction Bit Encoding Tables

This section provides various bit encoding tables for the instructions of the MIPS32 ISA.

Figure A-1 shows a sample encoding table and the instruction *opcode* field this table encodes. Bits 31..29 of the *opcode* field are listed in the leftmost columns of the table. Bits 28..26 of the *opcode* field are listed along the topmost rows of the table. Both decimal and binary values are given, with the first three bits designating the row, and the last three bits designating the column.

An instruction's encoding is found at the intersection of a row (bits 31..29) and column (bits 28..26) value. For instance, the *opcode* value for the instruction labelled EX1 is 33 (decimal, row and column), or 011011 (binary). Similarly, the *opcode* value for EX2 is 64 (decimal), or 110100 (binary).

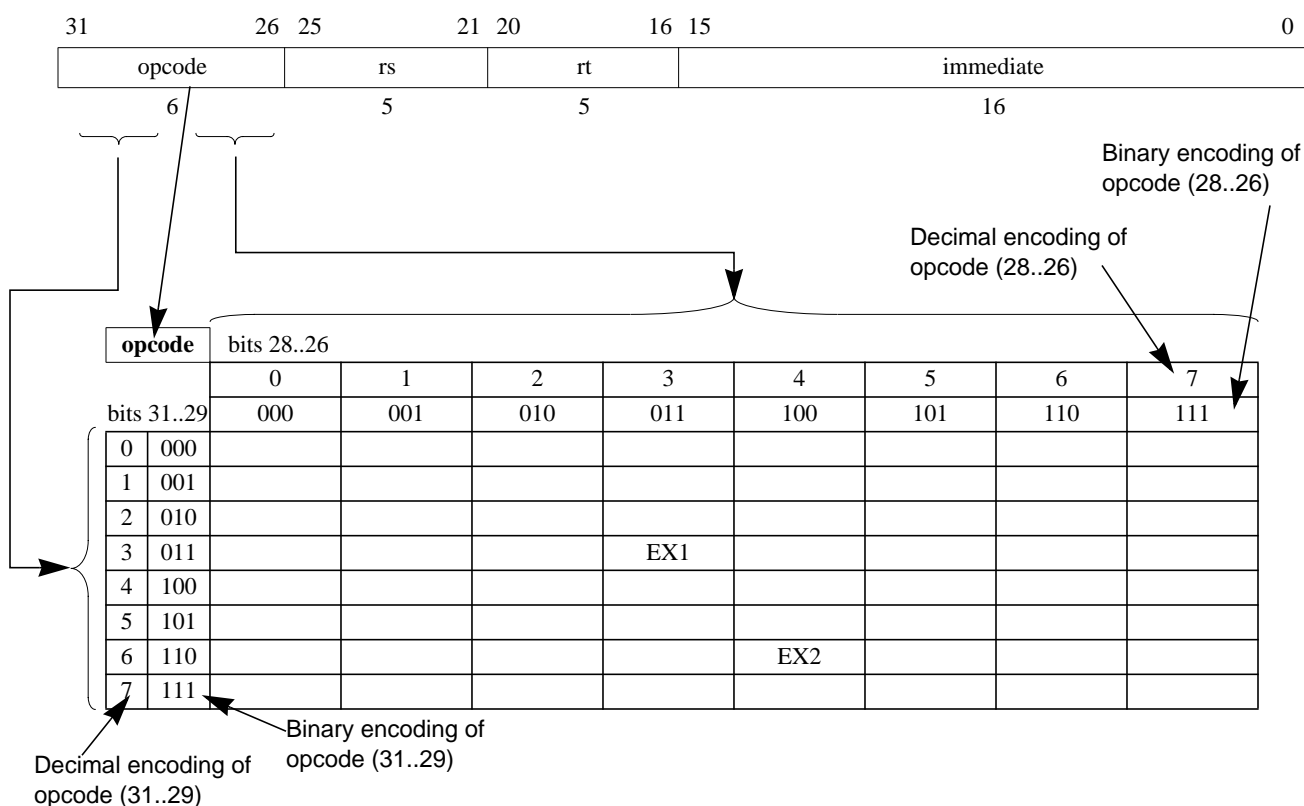


Figure A-1 Sample Bit Encoding Table

Tables A-2 through A-15 describe the encoding used for the MIPS32 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.
$\delta$	(Also <i>italic</i> 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.
$\beta$	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level. Executing such an instruction must cause a Reserved Instruction Exception.
$\theta$	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 ( <i>SPECIAL2</i> 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).
$\sigma$	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.



**Table A-1 Symbols Used in the Instruction Encoding Tables**

Symbol	Meaning
$\varepsilon$	Operation or field codes marked with this symbol are reserved for MIPS Application Specific Extensions. If the ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception.
$\phi$	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS32 ISA. Software should avoid using these operation or field codes.

**Table A-2 MIPS32 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	<i>SPECIAL</i> $\delta$	<i>REGIMM</i> $\delta$	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	<i>COP0</i> $\delta$	<i>COP1</i> $\delta$	<i>COP2</i> $\theta\delta$	<i>COP3</i> $\theta\delta$	BEQL $\phi$	BNEL $\phi$	BLEZL $\phi$	BGTZL $\phi$
3	011	$\beta$	$\beta$	$\beta$	$\beta$	<i>SPECIAL2</i> $\delta$	JALX $\varepsilon$	$\varepsilon$	*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	$\beta$
5	101	SB	SH	SWL	SW	$\beta$	$\beta$	SWR	CACHE
6	110	LL	LWC1	LWC2 $\theta$	PREF	$\beta$	LDC1	LDC2 $\theta$	$\beta$
7	111	SC	SWC1	SWC2 $\theta$	*	$\beta$	SDC1	SDC2 $\theta$	$\beta$

**Table A-3 MIPS32 *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	<i>MOVCI</i> $\delta$	SRL	SRA	SLLV	*	SRLV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	$\beta$	*	$\beta$	$\beta$
3	011	MULT	MULTU	DIV	DIVU	$\beta$	$\beta$	$\beta$	$\beta$
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	$\beta$	$\beta$	$\beta$	$\beta$
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	$\beta$	*	$\beta$	$\beta$	$\beta$	*	$\beta$	$\beta$

**Table A-4 MIPS32 *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 $\phi$	BGEZL $\phi$	*	*	*	*
1	01	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	10	BLTZAL	BGEZAL	BLTZALL $\phi$	BGEZALL $\phi$	*	*	*	*
3	11	*	*	*	*	*	*	*	*

**Table A-5 MIPS32 *SPECIAL2* Encoding of Function Field**

<b>function</b>		<i>bits 2..0</i>							
		0	1	2	3	4	5	6	7
<i>bits 5..3</i>		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	0	MSUB	MSUBU	0	0
1	001	0	0	0	0	0	0	0	0
2	010	0	0	0	0	0	0	0	0
3	011	0	0	0	0	0	0	0	0
4	100	CLZ	CLO	0	0	$\beta$	$\beta$	0	0
5	101	0	0	0	0	0	0	0	0
6	110	0	0	0	0	0	0	0	0
7	111	0	0	0	0	0	0	0	SDBBP $\sigma$

**Table A-6 MIPS32 *MOVCI* Encoding of tf Bit**

<b>tf</b>	<i>bit 16</i>	
	0	1
	MOVf	MOVt

**Table A-7 MIPS32 *COPz* Encoding of rs Field**

<b>rs</b>		<i>bits 23..21</i>							
		0	1	2	3	4	5	6	7
<i>bits 25..24</i>		000	001	010	011	100	101	110	111
0	00	MFCz	$\beta$	CFCz	*	MTCz	$\beta$	CTCz	*
1	01	BCz $\delta$	*	*	*	*	*	*	*
2	10	<i>CO <math>\delta</math></i>							
3	11								

**Table A-8 MIPS32 *COPz* Encoding of rt Field When rs=BCz**

<b>rt</b>	<i>bits 16</i>	
<i>bit 17</i>	0	1
0	BCzF	BCzT
1	BCzFL $\phi$	BCzTL $\phi$

**Table A-9 MIPS32 *COP0* Encoding of rs Field**

<b>rs</b>		<i>bits 23..21</i>							
		0	1	2	3	4	5	6	7
<i>bits 25..24</i>		000	001	010	011	100	101	110	111
0	00	MFC0	$\beta$	*	*	MTC0	$\beta$	*	*
1	01	*	*	*	*	*	*	*	*
2	10	<i>CO <math>\delta</math></i>							
3	11								

**Table A-10 MIPS32 *COP0* Encoding of Function Field When *rs=CO***

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	TLBR	TLBWI	*	*	*	TLBWR	*
1	001	TLBP	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	ERET	*	*	*	*	*	*	DERET $\sigma$
4	100	WAIT	*	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

**Table A-11 MIPS32 *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	$\beta$	CFC1	*	MTC1	$\beta$	CTC1	*
1	01	$BC1\ \delta$	$\epsilon$	$\epsilon\perp$	*	*	*	*	*
2	10	$S\ \delta$	$D\ \delta$	*	*	$W\ \delta$	$\beta$	$\beta$	*
3	11	*	*	*	*	*	*	*	*

**Table A-12 MIPS32 *COP1* Encoding of Function Field When *rs=S***

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	$\beta$	$\beta$	$\beta$	$\beta$	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	$MOVCF\ \delta$	MOVZ	MOVN	*	$\beta$	$\beta$	*
3	011	*	*	*	*	$\epsilon$	$\epsilon$	$\epsilon$	$\epsilon$
4	100	*	CVT.D	*	*	CVT.W	$\beta$	$\beta$	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7	111	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

**Table A-13 MIPS32 *COP1* Encoding of Function Field When *rs=D***

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	$\beta$	$\beta$	$\beta$	$\beta$	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	$MOVCF\ \delta$	MOVZ	MOVN	*	$\beta$	$\beta$	*
3	011	*	*	*	*	$\epsilon$	$\epsilon$	$\epsilon$	$\epsilon$
4	100	CVT.S	*	*	*	CVT.W	$\beta$	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7	111	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

**Table A-14 MIPS32 *COP1* Encoding of Function Field When *rs=W***

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	*	*	*	*	*	*	*
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	CVT.S	CVT.D	*	*	*	*	ε	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

**Table A-15 MIPS32 *COP1* Encoding of *tf* Bit When *rs=S, D, or PS*, Function=*MOVCF***

tf	bit 16	
	0	1
	MOV.F.fmt	MOV.T.fmt

### 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 table [Table A-11](#) above.

**Table 3-30 Floating Point Unit Instruction Format Encodings**

<i>fmt</i> field (bits 25..21 of COP1 opcode)		<i>fmt3</i> 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
17	11	1	1	D	Double	64	Floating Point
18..19	12..13	2..3	2..3	Reserved for future use by the architecture.			
20	14	4	4	W	Word	32	Fixed Point
21	15	5	5	L	Long	64	Fixed Point
22	16	6	6	PS	Paired Single	2 × 32	Floating Point
23	17	7	7	Reserved for future use by the architecture.			
24..31	18..1F	—	—	Reserved for future use by the architecture. Not available for <i>fmt3</i> encoding.			

## Revision History

In the left hand page margins of this document you may find vertical change bars to note the location of significant changes to this document since its last release. Significant changes are defined as those which you should take note of as you use the MIPS IP. Changes to correct grammar, spelling errors or similar may or may not be noted with change bars. Change bars will be removed for changes which are more than one revision old.

Please note: Limitations on the authoring tools make it difficult to place change bars on changes to figures. Change bars on figure titles are used to denote a potential change in the figure itself.

Revision	Date	Description
0.90	November 1, 2000	Internal review copy of reorganized and updated architecture documentation.
0.91	November 15, 2000	External review copy of reorganized and updated architecture documentation.
		Changes in this revision:
0.92	December 15, 2000	<ul style="list-style-type: none"> <li>• Correct sign in description of MSUBU.</li> <li>• Update JR and JALR instructions to reflect the changes required by MIPS16.</li> </ul>
0.95	March 12, 2001	Update for second external review release. Updated based on feedback from all reviews. <ul style="list-style-type: none"> <li>• Add missing optional select field syntax in mtc0/mfc0 instruction descriptions.</li> <li>• Correct the PREF instruction description to acknowledge that the PrepareForStore function does, in fact, modify architectural state.</li> <li>• To provide additional flexibility for Coprocessor 2 implementations, extend the <i>sel</i> field for DMFC0, DMTC0, MFC0, and MTC0 to be 8 bits.</li> <li>• Update the PREF instruction to note that it may not update the state of a locked cache line.</li> <li>• Remove obviously incorrect documentation in DIV and DIVU with regard to putting smaller numbers in register <i>rt</i>.</li> <li>• Fix the description for MFC2 to reflect data movement from the coprocessor 2 register to the GPR, rather than the other way around.</li> </ul>
1.00	August 29, 2002	<ul style="list-style-type: none"> <li>• Correct the pseudo code for LDC1, LDC2, SDC1, and SDC2 for a MIPS32 implementation to show the required word swapping.</li> <li>• 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.</li> <li>• Indicate that an Index Load Tag or Index Store Tag operation of the CACHE instruction must not cause a cache error exception.</li> <li>• 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.</li> <li>• Clean up the definitions of LL, SC, LLD, and SCD.</li> <li>• Add a warning that software should not use non-zero values of the stype field of the SYNC instruction.</li> <li>• Update the compatibility and subsetting rules to capture the current requirements.</li> </ul>

