



MIPS® Architecture For Programmers

Volume II-A: The MIPS64® Instruction Set

Document Number: MD00087
Revision 5.01
December 15, 2012

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

Copyright © 2001-2003,2005,2008-2012 MIPS Technologies Inc. All rights reserved.



Copyright © 2001-2003,2005,2008-2012 MIPS Technologies, Inc. All rights reserved.

Unpublished rights (if any) reserved under the copyright laws of the United States of America and other countries.

This document contains information that is proprietary to MIPS Technologies, Inc. ("MIPS Technologies"). Any copying, reproducing, modifying or use of this information (in whole or in part) that is not expressly permitted in writing by MIPS Technologies or an 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.

Any document provided in source format (i.e., in a modifiable form such as in FrameMaker or Microsoft Word format) is subject to use and distribution restrictions that are independent of and supplemental to any and all confidentiality restrictions. UNDER NO CIRCUMSTANCES MAY A DOCUMENT PROVIDED IN SOURCE FORMAT BE DISTRIBUTED TO A THIRD PARTY IN SOURCE FORMAT WITHOUT THE EXPRESS WRITTEN PERMISSION OF MIPS TECHNOLOGIES, INC.

MIPS Technologies 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 or 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. Except as expressly provided in any written license agreement from MIPS Technologies or an authorized third party, the furnishing of this document does not give recipient any license to any intellectual property rights, including any patent rights, that cover the information in this document.

The information contained in this document shall not be exported, reexported, transferred, or released, directly or indirectly, in violation of the law of any country or international law, regulation, treaty, Executive Order, statute, amendments or supplements thereto. Should a conflict arise regarding the export, reexport, transfer, or release of the information contained in this document, the laws of the United States of America shall be the governing law.

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 an authorized third party.

MIPS, MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPSr3, MIPS32, MIPS64, microMIPS32, microMIPS64, MIPS-3D, MIPS16, MIPS16e, MIPS-Based, MIPSSim, MIPSpro, MIPS-VERIFIED, Aptiv logo, microMIPS logo, MIPS Technologies logo, MIPS-VERIFIED logo, 4K, 4Kc, 4Km, 4Kp, 4KE, 4KEc, 4KEm, 4KEp, 4KS, 4KSc, 4KSd, M4K, M14K, 5K, 5Kc, 5Kf, 24K, 24Kc, 24Kf, 24KE, 24KEc, 24KEf, 34K, 34Kc, 34Kf, 74K, 74Kc, 74Kf, 1004K, 1004Kc, 1004Kf, 1074K, 1074Kc, 1074Kf, R3000, R4000, R5000, Aptiv, ASMACRO, Atlas, "At the core of the user experience.", BusBridge, Bus Navigator, CLAM, CorExtend, CoreFPGA, CoreLV, EC, FPGA View, FS2, FS2 FIRST SILICON SOLUTIONS logo, FS2 NAVIGATOR, HyperDebug, HyperJTAG, IASim, iFlowtrace, interAptiv, JALGO, Logic Navigator, Malta, MDMX, MED, MGB, microAptiv, microMIPS, OCI, PDtrace, the Pipeline, proAptiv, Pro Series, SEAD, SEAD-2, SmartMIPS, SOC-it, System Navigator, and YAMON are trademarks or registered trademarks of MIPS Technologies, Inc. in the United States and other countries. All other trademarks referred to herein are the property of their respective owners.

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

Template: nB1.02, Built with tags: 2B ARCH FPU_PS FPU_PSandARCH MIPS64

Contents

Chapter 1: About This Book	15
1.1: Typographical Conventions	15
1.1.1: Italic Text.....	16
1.1.2: Bold Text.....	16
1.1.3: Courier Text	16
1.2: UNPREDICTABLE and UNDEFINED	16
1.2.1: UNPREDICTABLE	16
1.2.2: UNDEFINED	17
1.2.3: UNSTABLE	17
1.3: Special Symbols in Pseudocode Notation.....	17
1.4: For More Information	20
Chapter 2: Guide to the Instruction Set	21
2.1: Understanding the Instruction Fields	21
2.1.1: Instruction Fields	23
2.1.2: Instruction Descriptive Name and Mnemonic.....	23
2.1.3: Format Field	23
2.1.4: Purpose Field	24
2.1.5: Description Field	24
2.1.6: Restrictions Field.....	24
2.1.7: Operation Field.....	25
2.1.8: Exceptions Field.....	25
2.1.9: Programming Notes and Implementation Notes Fields.....	26
2.2: Operation Section Notation and Functions	26
2.2.1: Instruction Execution Ordering.....	26
2.2.2: Pseudocode Functions.....	26
2.3: Op and Function Subfield Notation.....	36
2.4: FPU Instructions	36
Chapter 3: The MIPS64® Instruction Set	37
3.1: Compliance and Subsetting.....	37
3.2: Alphabetical List of Instructions	38
ABS(fmt	50
ADD.....	51
ADD.fmt.....	52
ADDI.....	53
ADDIU	54
ADDU	55
ALNV.PS	56
AND	58
ANDI	59
B	60
BAL.....	61
BC1F	62
BC1FL	64
BC1T	66
BC1TL	68

BC2F	70
BC2FL	71
BC2T	72
BC2TL	73
BEQ.....	74
BEQL.....	75
BGEZ.....	76
BGEZAL	77
BGEZALL	78
BGEZL.....	80
BGTZ.....	81
BGTZL.....	82
BLEZ	83
BLEZL	84
BLTZ.....	85
BLTZAL	86
BLTZALL	87
BLTZL.....	89
BNE	90
BNEL.....	91
BREAK	92
C.cond(fmt).....	93
CACHE	97
CACHEE	103
CEIL.L(fmt).....	110
CEIL.W(fmt).....	111
CFC1	112
CFC2	113
CLO	114
COP2.....	115
CLZ.....	116
CTC1	117
CTC2	119
CVT.D(fmt).....	120
CVT.L(fmt).....	121
CVT.PS.S	122
CVT.S(fmt).....	123
CVT.S.PL	124
CVT.S.PU	125
CVT.W(fmt).....	126
DADD	127
DADDI	128
DADDIU.....	129
DADDU.....	130
DCLO	131
DCLZ	132
DDIV	133
DDIVU	134
DERET	135
DEXT	136
DEXTM.....	138
DEXTU	140
DI.....	142

DINS	143
DINSM	145
DINSU	147
DIV	149
DIV.fmt	151
DIVU	152
DMFC0	153
DMFC1	154
DMFC2	155
DMTC0	156
DMTC1	157
DMTC2	158
DMULT	159
DMULTU	160
DROTR	161
DROTR32	162
DROTRV	163
DSBH	164
DSHD	165
DSLL	166
DSLL32	167
DSLLV	168
DSRA	169
DSRA32	170
DSRAV	171
DSRL	172
DSRL32	173
DSRLV	174
DSUB	175
DSUBU	176
EHB	177
EI	178
ERET	179
EXT	180
FLOOR.L fmt	182
FLOOR.W fmt	183
INS	184
J	186
JAL	187
JALR	188
JALR.HB	190
JALX	193
JR	194
JR.HB	196
LB	198
LBE	200
LBU	201
LBUE	203
LD	204
LDC1	205
LDC2	206
LDL	207
LDR	209

LDXC1.....	211
LH.....	212
LHE	214
LHU	215
LHUE.....	217
LL	218
LLE	220
LLD.....	222
LUI.....	223
LUXC1.....	224
LW	225
LWC1	226
LWC2	227
LWE.....	228
LWL.....	229
LWLE.....	231
LWR	233
LWRE	236
LWU	239
LWXC1	240
MADD.....	241
MADD(fmt	242
MADDU	243
MFC0.....	244
MFC1.....	245
MFC2.....	246
MFHC1	247
MFHC2	248
MFHI.....	249
MFLO	250
MOV(fmt	251
MOVF	252
MOVF(fmt.....	253
MOVN	255
MOVN(fmt.....	256
MOVT	257
MOVT(fmt.....	258
MOVZ	260
MOVZ(fmt.....	261
MSUB	262
MSUB(fmt.....	263
MSUBU	264
MTC0.....	265
MTC1.....	266
MTC2.....	267
MTHC1	268
MTHC2	269
MTHI.....	270
MTLO	271
MUL.....	272
MUL(fmt.....	273
MULT.....	274
MULTU	275

NEG(fmt)	276
NMADD(fmt)	277
NMSUB(fmt)	279
NOP	281
NOR	282
OR	283
ORI	284
PAUSE	285
PLL.PS	287
PLU.PS	288
PREF	289
PREFE	292
PREFIX	295
PUL.PS	296
PUU.PS	297
RDHWR	298
RDPGPR	300
RECIP(fmt)	301
ROTR	302
ROTRV	303
ROUND.L(fmt)	304
ROUND.W(fmt)	305
RSQRT(fmt)	306
SB	307
SBE	308
SC	309
SCE	313
SCD	316
SD	318
SDBBP	319
SDC1	320
SDC2	321
SDL	322
SDR	324
SDXC1	326
SEB	327
SEH	328
SH	330
SHE	332
SLL	333
SLLV	334
SLT	335
SLTI	336
SLTIU	337
SLTU	338
SQRT(fmt)	339
SRA	340
SRAV	341
SRL	342
SRLV	343
SSNOP	344
SUB	345
SUB(fmt)	346

SUBU	347
SUXC1	348
SW.....	349
SWC1	350
SWC2	351
SWE	353
SWL.....	354
SWLE	356
SWR	358
SWRE.....	360
SWXC1.....	362
SYNC	363
SYNCI	368
SYSCALL.....	370
TEQ.....	371
TEQI	372
TGE	373
TGEI	374
TGEIU	375
TGEU	376
TLBINV.....	378
TLBINVF.....	380
TLBP	382
TLBR	383
TLBWI	385
TLBWR	387
TLT	389
TLTI.....	390
TLTIU	391
TLTU	392
TNE	393
TNEI	394
TRUNC.L fmt.....	395
TRUNC.W fmt.....	396
WAIT	397
WRPGPR	398
WSBH.....	399
XOR.....	400
XORI.....	401
Appendix A: Instruction Bit Encodings	403
A.1: Instruction Encodings and Instruction Classes	403
A.2: Instruction Bit Encoding Tables.....	403
A.3: Floating Point Unit Instruction Format Encodings	412
Appendix B: Revision History	413

Figures

Figure 2.1: Example of Instruction Description	22
Figure 2.2: Example of Instruction Fields.....	23
Figure 2.3: Example of Instruction Descriptive Name and Mnemonic	23
Figure 2.4: Example of Instruction Format.....	23
Figure 2.5: Example of Instruction Purpose	24
Figure 2.6: Example of Instruction Description	24
Figure 2.7: Example of Instruction Restrictions.....	25
Figure 2.8: Example of Instruction Operation.....	25
Figure 2.9: Example of Instruction Exception.....	25
Figure 2.10: Example of Instruction Programming Notes	26
Figure 2.11: COP_LW Pseudocode Function	27
Figure 2.12: COP_LD Pseudocode Function.....	27
Figure 2.13: COP_SW Pseudocode Function	27
Figure 2.14: COP_SD Pseudocode Function	28
Figure 2.15: CoprocessorOperation Pseudocode Function	28
Figure 2.16: AddressTranslation Pseudocode Function	28
Figure 2.17: LoadMemory Pseudocode Function	29
Figure 2.18: StoreMemory Pseudocode Function.....	29
Figure 2.19: Prefetch Pseudocode Function.....	30
Figure 2.20: SyncOperation Pseudocode Function	31
Figure 2.21: ValueFPR Pseudocode Function.....	31
Figure 2.22: StoreFPR Pseudocode Function	32
Figure 2.23: CheckFPEexception Pseudocode Function	33
Figure 2.24: FPCConditionCode Pseudocode Function	33
Figure 2.25: SetFPCConditionCode Pseudocode Function	33
Figure 2.26: SignalException Pseudocode Function	34
Figure 2.27: SignalDebugBreakpointException Pseudocode Function.....	34
Figure 2.28: SignalDebugModeBreakpointException Pseudocode Function.....	34
Figure 2.29: NullifyCurrentInstruction PseudoCode Function	35
Figure 2.30: JumpDelaySlot Pseudocode Function	35
Figure 2.31: NotWordValue Pseudocode Function.....	35
Figure 2.32: PolyMult Pseudocode Function	35
Figure 3.1: Example of an ALNV.PS Operation.....	56
Figure 3.2: Usage of Address Fields to Select Index and Way.....	97
Figure 3.3: Usage of Address Fields to Select Index and Way.....	103
Figure 3.4: Operation of the DEXT Instruction	136
Figure 3.5: Operation of the DEXTM Instruction	138
Figure 3.6: Operation of the DEXTU Instruction	140
Figure 3.7: Operation of the DINS Instruction	143
Figure 3.8: Operation of the DINSM Instruction	145
Figure 3.9: Operation of the DINSU Instruction	147
Figure 3.10: Operation of the EXT Instruction	180
Figure 3.11: Operation of the INS Instruction	184
Figure 3.12: Unaligned Doubleword Load Using LDL and LDR.....	207
Figure 3.13: Bytes Loaded by LDL Instruction.....	208
Figure 3.14: Unaligned Doubleword Load Using LDR and LDL.....	209
Figure 3.15: Bytes Loaded by LDR Instruction	210

Figure 3.16: Unaligned Word Load Using LWL and LWR.....	229
Figure 3.17: Bytes Loaded by LWL Instruction	230
Figure 3.18: Unaligned Word Load Using LWLE and LWRE.....	231
Figure 3.19: Bytes Loaded by LWLE Instruction.....	232
Figure 3.20: Unaligned Word Load Using LWL and LWR.....	233
Figure 3.21: Bytes Loaded by LWR Instruction.....	234
Figure 3.22: Unaligned Word Load Using LWLE and LWRE.....	236
Figure 3.23: Bytes Loaded by LWRE Instruction	237
Figure 3.24: Unaligned Doubleword Store With SDL and SDR	322
Figure 3.25: Bytes Stored by an SDL Instruction.....	323
Figure 3.26: Unaligned Doubleword Store With SDR and SDL	324
Figure 3.27: Bytes Stored by an SDR Instruction	325
Figure 3.28: Unaligned Word Store Using SWL and SWR	354
Figure 3.29: Bytes Stored by an SWL Instruction	355
Figure 3.30: Unaligned Word Store Using SWLE and SWRE	356
Figure 3.31: Bytes Stored by an SWLE Instruction.....	357
Figure 3.32: Unaligned Word Store Using SWR and SWL	358
Figure 3.33: Bytes Stored by SWR Instruction.....	359
Figure 3.34: Unaligned Word Store Using SWRE and SWLE	360
Figure 3.35: Bytes Stored by SWRE Instruction	361
Figure A.1: Sample Bit Encoding Table	404

Tables

Table 1.1: Symbols Used in Instruction Operation Statements.....	17
Table 2.1: AccessLength Specifications for Loads/Stores	30
Table 3.1: CPU Arithmetic Instructions	38
Table 3.2: CPU Branch and Jump Instructions	39
Table 3.3: CPU Instruction Control Instructions	40
Table 3.4: CPU Load, Store, and Memory Control Instructions	40
Table 3.5: CPU Logical Instructions.....	42
Table 3.6: CPU Insert/Extract Instructions	42
Table 3.7: CPU Move Instructions	43
Table 3.8: CPU Shift Instructions	43
Table 3.9: CPU Trap Instructions.....	44
Table 3.10: Obsolete CPU Branch Instructions	44
Table 3.11: FPU Arithmetic Instructions.....	45
Table 3.12: FPU Branch Instructions	45
Table 3.13: FPU Compare Instructions.....	46
Table 3.14: FPU Convert Instructions	46
Table 3.15: FPU Load, Store, and Memory Control Instructions	46
Table 3.16: FPU Move Instructions.....	47
Table 3.17: Obsolete FPU Branch Instructions	47
Table 3.18: Coprocessor Branch Instructions	48
Table 3.19: Coprocessor Execute Instructions	48
Table 3.20: Coprocessor Load and Store Instructions..	48
Table 3.21: Coprocessor Move Instructions.....	48
Table 3.22: Obsolete Coprocessor Branch Instructions	49
Table 3.23: Privileged Instructions	49
Table 3.24: EJTAG Instructions	49
Table 3.25: FPU Comparisons Without Special Operand Exceptions	94
Table 3.26: FPU Comparisons With Special Operand Exceptions for QNaNs	95
Table 3.27: Usage of Effective Address.....	97
Table 3.28: Encoding of Bits[17:16] of CACHE Instruction	98
Table 3.29: Encoding of Bits [20:18] of the CACHE Instruction.....	99
Table 3.30: Usage of Effective Address.....	103
Table 3.31: Encoding of Bits[17:16] of CACHEE Instruction.....	104
Table 3.32: Encoding of Bits [20:18] of the CACHEE Instruction.....	105
Table 3.33: Values of <i>hint</i> Field for PREF Instruction	289
Table 3.34: Values of <i>hint</i> Field for PREFE Instruction.....	293
Table 3.35: RDHWR Register Numbers	298
Table 3.36: Encodings of the Bits[10:6] of the SYNC instruction; the SType Field.....	365
Table A.1: Symbols Used in the Instruction Encoding Tables	404
Table A.2: MIPS64 Encoding of the Opcode Field	405
Table A.3: MIPS64 SPECIAL Opcode Encoding of Function Field.....	406
Table A.4: MIPS64 REG/IMM Encoding of <i>rt</i> Field	406
Table A.5: MIPS64 SPECIAL2 Encoding of Function Field	406
Table A.6: MIPS64 SPECIAL3 Encoding of Function Field for Release 2 of the Architecture.....	407
Table A.7: MIPS64 MOVC/ Encoding of <i>tf</i> Bit	407
Table A.8: MIPS64 SRL Encoding of Shift/Rotate	407
Table A.9: MIPS64 SRLV Encoding of Shift/Rotate.....	407

Table A.10: MIPS64 <i>DSRLV</i> Encoding of Shift/Rotate	408
Table A.11: MIPS64 <i>DSRL</i> Encoding of Shift/Rotate.....	408
Table A.12: MIPS64 <i>DSRL32</i> Encoding of Shift/Rotate.....	408
Table A.13: MIPS64 <i>BSHFL</i> and <i>DBSHFL</i> Encoding of <i>sa</i> Field.....	408
Table A.14: MIPS64 <i>COP0</i> Encoding of <i>rs</i> Field	409
Table A.15: MIPS64 <i>COP0</i> Encoding of Function Field When <i>rs</i> = <i>CO</i>	409
Table A.16: MIPS64 <i>COP1</i> Encoding of <i>rs</i> Field	409
Table A.17: MIPS64 <i>COP1</i> Encoding of Function Field When <i>rs</i> = <i>S</i>	410
Table A.18: MIPS64 <i>COP1</i> Encoding of Function Field When <i>rs</i> = <i>D</i>	410
Table A.19: MIPS64 <i>COP1</i> Encoding of Function Field When <i>rs</i> = <i>W</i> or <i>L</i>	410
Table A.20: MIPS64 <i>COP1</i> Encoding of Function Field When <i>rs</i> = <i>PS</i>	411
Table A.21: MIPS64 <i>COP1</i> Encoding of <i>tf</i> Bit When <i>rs</i> = <i>S</i> , <i>D</i> , or <i>PS</i> , Function= <i>MOVCF</i>	411
Table A.22: MIPS64 <i>COP2</i> Encoding of <i>rs</i> Field	411
Table A.23: MIPS64 <i>COP1X</i> Encoding of Function Field	411
Table A.24: Floating Point Unit Instruction Format Encodings.....	412

About This Book

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

- Volume I-A describes conventions used throughout the document set, and provides an introduction to the MIPS64® Architecture
- Volume I-B describes conventions used throughout the document set, and provides an introduction to the microMIPS64™ Architecture
- Volume II-A provides detailed descriptions of each instruction in the MIPS64® instruction set
- Volume II-B provides detailed descriptions of each instruction in the microMIPS64™ instruction set
- Volume III describes the MIPS64® and microMIPS64™ Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS® processor implementation
- Volume IV-a describes the MIPS16e™ Application-Specific Extension to the MIPS64® Architecture. Beginning with Release 3 of the Architecture, microMIPS is the preferred solution for smaller code size.
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64® Architecture and microMIPS64™. With Release 5 of the Architecture, MDMX is deprecated. MDMX and MSA can not be implemented at the same time.
- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS® Architecture
- Volume IV-d describes the SmartMIPS® Application-Specific Extension to the MIPS32® Architecture and the microMIPS32™ Architecture and is not applicable to the MIPS64® document set nor the microMIPS64™ document set.
- Volume IV-e describes the MIPS® DSP Module to the MIPS® Architecture
- Volume IV-f describes the MIPS® MT Module to the MIPS® Architecture
- Volume IV-h describes the MIPS® MCU Application-Specific Extension to the MIPS® Architecture
- Volume IV-i describes the MIPS® Virtualization Module to the MIPS® Architecture
- Volume IV-j describes the MIPS® SIMD Architecture Module to the MIPS® Architecture

1.1 Typographical Conventions

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

1.1.1 Italic Text

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

1.1.2 Bold Text

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

1.1.3 Courier Text

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

1.2 UNPREDICTABLE and UNDEFINED

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

1.2.1 UNPREDICTABLE

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

UNPREDICTABLE results or operations have several implementation restrictions:

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

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

1.2.2 UNDEFINED

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

UNDEFINED operations or behavior has one implementation restriction:

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

1.2.3 UNSTABLE

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

UNSTABLE values have one implementation restriction:

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

1.3 Special Symbols in Pseudocode Notation

In this book, algorithmic descriptions of an operation are described as pseudocode in a high-level language notation resembling Pascal. Special symbols used in the pseudocode notation are listed in [Table 1.1](#).

Table 1.1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
\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
$b\#n$	A constant value n in base b . For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10.
0bn	A constant value n in base 2. For instance 0b100 represents the binary value 100 (decimal 4).
0xn	A constant value n in base 16. For instance 0x100 represents the hexadecimal value 100 (decimal 256).
$x_{y..z}$	Selection of bits y through z of bit string x . Little-endian bit notation (rightmost bit is 0) is used. If y is less than z , this expression is an empty (zero length) bit string.
$+, -$	2's complement or floating point arithmetic: addition, subtraction

About This Book

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
\ast, \times	2's complement or floating point multiplication (both used for either)
div	2's complement integer division
mod	2's complement modulo
/	Floating point division
<	2's complement less-than comparison
>	2's complement greater-than comparison
\leq	2's complement less-than or equal comparison
\geq	2's complement greater-than or equal comparison
nor	Bitwise logical NOR
xor	Bitwise logical XOR
and	Bitwise logical AND
or	Bitwise logical OR
not	Bitwise inversion
$\&\&$	Logical (non-Bitwise) AND
\ll	Logical Shift left (shift in zeros at right-hand-side)
\gg	Logical Shift right (shift in zeros at left-hand-side)
GPRLEN	The length in bits (32 or 64) of the CPU general-purpose registers
$GPR[x]$	CPU general-purpose register x . The content of $GPR[0]$ is always zero. In Release 2 of the Architecture, $GPR[x]$ is a short-hand notation for $SGPR[SRSCtl_{CSS}, x]$.
$SGPR[s,x]$	In Release 2 of the Architecture and subsequent releases, multiple copies of the CPU general-purpose registers may be implemented. $SGPR[s,x]$ refers to GPR set s , register x .
$FPR[x]$	Floating Point operand register x
$FCC[CC]$	Floating Point condition code CC. $FCC[0]$ has the same value as $COC[1]$.
$FPR[x]$	Floating Point (Coprocessor unit 1), general register x
$CPR[z,x,s]$	Coprocessor unit z , general register x , select s
$CP2CPR[x]$	Coprocessor unit 2, general register x
$CCR[z,x]$	Coprocessor unit z , control register x
$CP2CCR[x]$	Coprocessor unit 2, control register x
$COC[z]$	Coprocessor unit z condition signal
$Xlat[x]$	Translation of the MIPS16e GPR number x into the corresponding 32-bit GPR number
BigEndianMem	Endian mode as configured at chip reset (0 → Little-Endian, 1 → Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution.
BigEndianCPU	The endianness for load and store instructions (0 → Little-Endian, 1 → Big-Endian). In User mode, this endianness may be switched by setting the RE bit in the $Status$ register. Thus, BigEndianCPU may be computed as ($BigEndianMem \text{ XOR } ReverseEndian$).
ReverseEndian	Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the RE bit of the $Status$ register. Thus, ReverseEndian may be computed as (SR_{RE} and User mode).

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning						
<i>LLbit</i>	Bit of virtual state used to specify operation for instructions that provide atomic read-modify-write. <i>LLbit</i> is set when a linked load occurs and is tested by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.						
I: , I+n: , I-n:	This occurs as a prefix to <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 I . Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction I , in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled I+1 . The effect of pseudocode statements for the current instruction labelled I+1 appears to occur “at the same time” as the effect of pseudocode statements labeled I for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur “at the same time,” there is no defined order. Programs must not depend on a particular order of evaluation between such sections.						
PC	The <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 PC during an instruction time. If no value is assigned to PC during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the PC during the instruction time of the instruction in the branch delay slot. In the MIPS Architecture, the PC value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. The PC value contains a full 64-bit address all of which are significant during a memory reference.						
ISA Mode	In processors that implement the MIPS16e Application Specific Extension or the microMIPS base architectures, the <i>ISA Mode</i> is a single-bit register that determines in which mode the processor is executing, as follows: <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>The processor is executing 32-bit MIPS instructions</td> </tr> <tr> <td>1</td> <td>The processor is executing MIIPS16e or microMIPS instructions</td> </tr> </tbody> </table> In the MIPS Architecture, the ISA Mode value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the ISA Mode into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception.	Encoding	Meaning	0	The processor is executing 32-bit MIPS instructions	1	The processor is executing MIIPS16e or microMIPS instructions
Encoding	Meaning						
0	The processor is executing 32-bit MIPS instructions						
1	The processor is executing MIIPS16e or microMIPS instructions						
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.						
SEGBITS	The number of virtual address bits implemented in a segment of the address space is represented by the symbol SEGBITS. As such, if 40 virtual address bits are implemented in a segment, the size of the segment is $2^{\text{SEGBITS}} = 2^{40}$ bytes.						

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
FP32RegistersMode	Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32 Release 1, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, (and optionally in MIPS32 Release2 and MIPSr3) the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR. In MIPS32 Release 1 implementations, FP32RegistersMode is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case FP32RegisterMode is computed from the FR bit in the <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. The value of FP32RegistersMode is computed from the FR bit in the <i>Status</i> register.
InstructionInBranchDelaySlot	Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the <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>

For comments or questions on the MIPS64® Architecture or this document, send Email to support@mips.com.

Guide to the Instruction Set

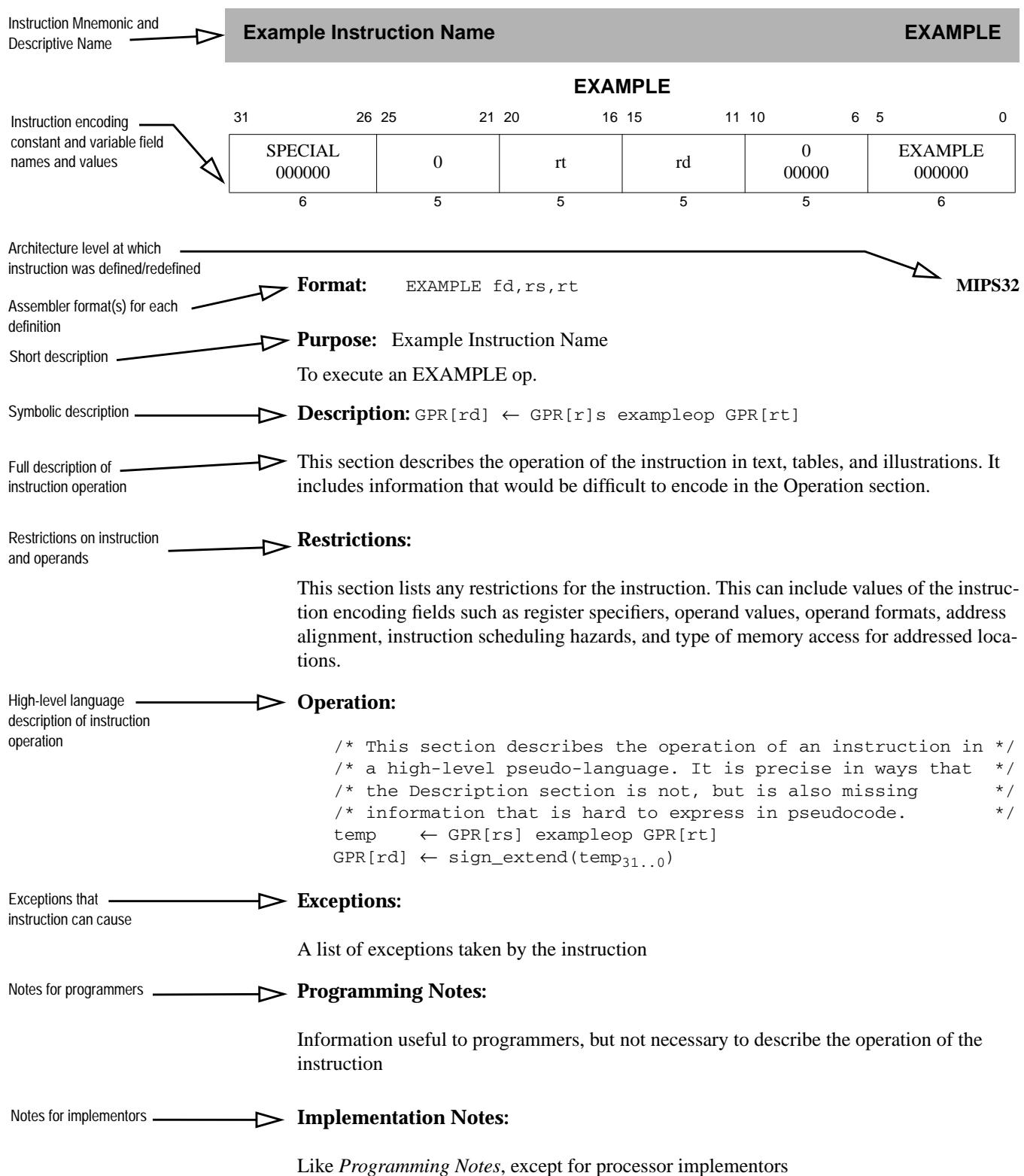
This chapter provides a detailed guide to understanding the instruction descriptions, which are listed in alphabetical order in the tables at the beginning of the next chapter.

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 23
- “Instruction Descriptive Name and Mnemonic” on page 23
- “Format Field” on page 23
- “Purpose Field” on page 24
- “Description Field” on page 24
- “Restrictions Field” on page 24
- “Operation Field” on page 25
- “Exceptions Field” on page 25
- “Programming Notes and Implementation Notes Fields” on page 26

Figure 2.1 Example of Instruction Description

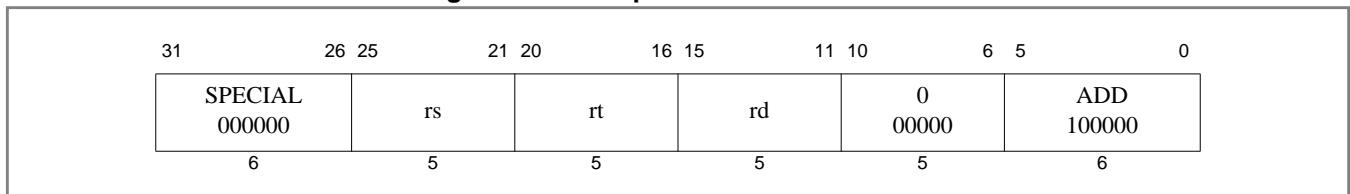


2.1.1 Instruction Fields

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

- The values of constant fields and the *opcode* names are listed in uppercase (SPECIAL and ADD in Figure 2.2). Constant values in a field are shown in binary below the symbolic or hexadecimal value.
- All variable fields are listed with the lowercase names used in the instruction description (*rs*, *rt*, and *rd* in Figure 2.2).
- Fields that contain zeros but are not named are unused fields that are required to be zero (bits 10:6 in Figure 2.2). If such fields are set to non-zero values, the operation of the processor is **UNPREDICTABLE**.

Figure 2.2 Example of Instruction Fields



2.1.2 Instruction Descriptive Name and Mnemonic

The instruction descriptive name and mnemonic are printed as page headings for each instruction, as shown in Figure 2.3.

Figure 2.3 Example of Instruction Descriptive Name and Mnemonic



2.1.3 Format Field

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

Figure 2.4 Example of Instruction Format



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.

Figure 2.5 Example of Instruction Purpose

Purpose: Add Word

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

2.1.5 Description Field

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

Figure 2.6 Example of Instruction Description

Description: GPR[rd] \leftarrow GPR[rs] + GPR[rt]

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

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

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

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

2.1.6 Restrictions Field

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

- Valid values for instruction fields (for example, see floating point [ADD.fmt](#))
- ALIGNMENT requirements for memory addresses (for example, see [LW](#))
- Valid values of operands (for example, see [DADD](#))

- Valid operand formats (for example, see floating point [ADD\(fmt\)](#))
- Order of instructions necessary to guarantee correct execution. These ordering constraints avoid pipeline hazards for which some processors do not have hardware interlocks (for example, see [MUL](#)).
- Valid memory access types (for example, see [LL/SC](#))

Figure 2.7 Example of Instruction Restrictions**Restrictions:**

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

2.1.7 Operation Field

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

Figure 2.8 Example of Instruction Operation**Operation:**

```

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

```

See [2.2 “Operation Section Notation and Functions” on page 26](#) 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.

Figure 2.9 Example of Instruction Exception**Exceptions:**

Integer Overflow

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

2.1.9 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.

Figure 2.10 Example of Instruction Programming Notes

Programming Notes:

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

2.2 Operation Section Notation and Functions

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

This section presents information about the following topics:

- “Instruction Execution Ordering” on page 26
- “Pseudocode Functions” on page 26

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 26
- “Memory Operation Functions” on page 28
- “Floating Point Functions” on page 31
- “Miscellaneous Functions” on page 34

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*.

Figure 2.11 COP_LW Pseudocode Function

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

    /* Coprocessor-dependent action */

endfunction COP_LW
```

COP_LD

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

Figure 2.12 COP_LD Pseudocode Function

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

    /* Coprocessor-dependent action */

endfunction COP_LD
```

COP_SW

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

Figure 2.13 COP_SW Pseudocode Function

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

    /* Coprocessor-dependent action */

endfunction COP_SW
```

COP_SD

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

Figure 2.14 COP_SD Pseudocode Function

```

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

    /* Coprocessor-dependent action */

endfunction COP_SD

```

CoprocessorOperation

The CoprocessorOperation function performs the specified Coprocessor operation.

Figure 2.15 CoprocessorOperation Pseudocode Function

```

CoprocessorOperation (z, cop_fun)

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

/* Transmit the cop_fun value to coprocessor z */

endfunction CoprocessorOperation

```

2.2.2.2 Memory Operation Functions

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

In the *Operation* pseudocode for load and store operations, the following functions summarize the handling of virtual addresses and the access of physical memory. The size of the data item to be loaded or stored is passed in the *AccessLength* field. The valid constant names and values are shown in Table 2.1. The bytes within the addressed unit of memory (word for 32-bit processors or doubleword for 64-bit processors) that are used can be determined directly from the *AccessLength* and the two or three low-order bits of the address.

AddressTranslation

The AddressTranslation function translates a virtual address to a physical address and its cacheability and coherency attribute, describing the mechanism used to resolve the memory reference.

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

Figure 2.16 AddressTranslation Pseudocode Function

```

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

/* pAddr: physical address */
/* CCA:   Cacheability&Coherency Attribute, the method used to access caches */

```

```

/* and memory and resolve the reference */

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

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

endfunction AddressTranslation

```

LoadMemory

The LoadMemory function loads a value from memory.

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

Figure 2.17 LoadMemory Pseudocode Function

```

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

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

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

endfunction LoadMemory

```

StoreMemory

The StoreMemory function stores a value to memory.

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

Figure 2.18 StoreMemory Pseudocode Function

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

```

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

endfunction StoreMemory

```

Prefetch

The Prefetch function prefetches data from memory.

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

Figure 2.19 Prefetch Pseudocode Function

```

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

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

endfunction Prefetch

```

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

Table 2.1 AccessLength Specifications for Loads/Stores

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

SyncOperation

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

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

Figure 2.20 SyncOperation Pseudocode Function

```
SyncOperation(stype)

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

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

endfunction SyncOperation
```

2.2.2.3 Floating Point Functions

The pseudocode shown 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.

Figure 2.21 ValueFPR Pseudocode Function

```
value ← ValueFPR(fpr, fmt)

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

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

case fmt of
    S, W, UNINTERPRETED_WORD:
        valueFPR ← UNPREDICTABLE32 || FPR[fpr]31..0

    D, UNINTERPRETED_DOUBLEWORD:
        if (FP32RegistersMode = 0)
            if (fpr0 ≠ 0) then
                valueFPR ← UNPREDICTABLE
            else
                valueFPR ← FPR[fpr+1]31..0 || FPR[fpr]31..0
            endif
        else
            valueFPR ← FPR[fpr]
        endif

    L, PS, OB, QH:
        if (FP32RegistersMode = 0) then
            valueFPR ← UNPREDICTABLE
```

Guide to the Instruction Set

```
        else
            valueFPR ← FPR[fpr]
        endif

        DEFAULT:
            valueFPR ← UNPREDICTABLE

    endcase
endfunction ValueFPR
```

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

StoreFPR

Figure 2.22 StoreFPR Pseudocode Function

```
StoreFPR (fpr, fmt, value)

/* fpr: The FPR number */
/* fmt: The format of the data, one of: */
/*      S, D, W, L, PS, */
/*      OB, QH, */
/*      UNINTERPRETED_WORD, */
/*      UNINTERPRETED_DOUBLEWORD */
/* value: The 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] ← UNPREDICTABLE32 || value31..0

    D, UNINTERPRETED_DOUBLEWORD:
        if (FP32RegistersMode = 0)
            if (fpr0 ≠ 0) then
                UNPREDICTABLE
            else
                FPR[fpr] ← UNPREDICTABLE32 || value31..0
                FPR[fpr+1] ← UNPREDICTABLE32 || value63..32
            endif
        else
            FPR[fpr] ← value
        endif

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

endcase
```

```
endfunction StoreFPR
```

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

CheckFPEException

Figure 2.23 CheckFPEException Pseudocode Function

```
CheckFPEException()

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

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

endfunction CheckFPEException
```

FPCConditionCode

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

Figure 2.24 FPCConditionCode Pseudocode Function

```
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 FPCConditionCode
```

SetFPCConditionCode

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

Figure 2.25 SetFPCConditionCode Pseudocode Function

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

endfunction SetFPCConditionCode
```

2.2.2.4 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

SignalException

The SignalException function signals an exception condition.

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

Figure 2.26 SignalException Pseudocode Function

```
SignalException(Exception, argument)

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

endfunction SignalException
```

SignalDebugBreakpointException

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

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

Figure 2.27 SignalDebugBreakpointException Pseudocode Function

```
SignalDebugBreakpointException()

endfunction SignalDebugBreakpointException
```

SignalDebugModeBreakpointException

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

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

Figure 2.28 SignalDebugModeBreakpointException Pseudocode Function

```
SignalDebugModeBreakpointException()

endfunction SignalDebugModeBreakpointException
```

NullifyCurrentInstruction

The NullifyCurrentInstruction function nullifies the current instruction.

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

Figure 2.29 NullifyCurrentInstruction PseudoCode Function

```
NullifyCurrentInstruction()
endfunction NullifyCurrentInstruction
```

JumpDelaySlot

The JumpDelaySlot function is used in the pseudocode for the PC-relative instructions in the MIPS16e ASE. The function returns TRUE if the instruction at *vAddr* is executed in a jump delay slot. A jump delay slot always immediately follows a JR, JAL, JALR, or JALX instruction.

Figure 2.30 JumpDelaySlot Pseudocode Function

```
JumpDelaySlot (vAddr)
/* vAddr:Virtual address */
endfunction JumpDelaySlot
```

NotWordValue

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

Figure 2.31 NotWordValue Pseudocode Function

```
result ← NotWordValue(value)

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

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

NotWordValue ← value63..32 ≠ (value31)32

endfunction NotWordValue
```

PolyMult

The PolyMult function multiplies two binary polynomial coefficients.

Figure 2.32 PolyMult Pseudocode Function

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

PolyMult ← temp

endfunction PolyMult
```

2.3 Op and Function Subfield Notation

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

2.4 FPU Instructions

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

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

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

See “Op and Function Subfield Notation” on page 36 for a description of the *op* and *function* subfields.

The MIPS64® Instruction Set

3.1 Compliance and Subsetting

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

Note: The use of COP3 as a customizable coprocessor has been removed in the Release 2 of the MIPS64 architecture. The use of the COP3 is now reserved for the future extension of the architecture.

The instruction set subsetting rules are as follows:

- All non-privileged (does not need access to Coprocessor 0) CPU (non-FPU) instructions must be implemented - no subsetting is allowed (unless described in this list).
- The FPU and related support instructions, including the MOVF and MOVT CPU instructions, may be omitted. Software may determine if an FPU is implemented by checking the state of the FP bit in the *Config1* CP0 register. If the FPU is implemented, the paired single (PS) format is optional. Software may determine which FPU data types are implemented by checking the appropriate bit in the *FIR* CP1 register. The following allowable FPU subsets are compliant with the MIPS64 architecture:
 - No FPU
 - FPU with S, D, W, and L formats and all supporting instructions
 - FPU with S, D, PS, W, and L formats and all supporting instructions
- Coprocessor 2 is optional and may be omitted. Software may determine if Coprocessor 2 is implemented by checking the state of the C2 bit in the *Config1* CP0 register. If Coprocessor 2 is implemented, the Coprocessor 2 interface instructions (BC2, CFC2, COP2, CTC2, DMFC2, DMTC2, LDC2, LWC2, MFC2, MTC2, SDC2, and SWC2) may be omitted on an instruction-by-instruction basis.
- 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 MMU is not implemented, the TLB related instructions can be subsetted out. Software may determine the type of the MMU by checking the MT field in the *Config* CP0 register.
- Instruction, CP0 Register, and CP1 Control Register fields that are marked “Reserved” or shown as “0” in the description of that field are reserved for future use by the architecture and are not available to implementations. Implementations may only use those fields that are explicitly reserved for implementation dependent use.

- Supported Modules and ASEs are optional and may be subsetted out. If most cases, software may determine if a supported Module/ASE is implemented by checking the appropriate bit in the *Config1* or *Config3* or *Config4* CP0 register. If they are implemented, they must implement the entire ISA applicable to the component, or implement subsets that are approved by the ASE specifications.
- EJTAG is optional and may be subsetted out. If it is implemented, it must implement only those subsets that are approved by the EJTAG specification. If EJTAG is not implemented, the EJTAG instructions (SDBBP and DERET) can be subsetted out.
- The JALX instruction is only implemented when there are other instruction sets available on the device (microMIPS or MIPS16e).
- EVA load/store (LWE, LHE, LHUE, LBE, LBUE, SWE, SHE, SBE) instructions are optional.
- If any instruction is subsetted out based on the rules above, an attempt to execute that instruction must cause the appropriate exception (typically Reserved Instruction or Coprocessor Unusable).

3.2 Alphabetical List of Instructions

Table 3.1 through Table 3.24 provide a list of instructions grouped by category. Individual instruction descriptions follow the tables, arranged in alphabetical order.

Table 3.1 CPU Arithmetic Instructions

Mnemonic	Instruction
ADD	Add Word
ADDI	Add Immediate Word
ADDIU	Add Immediate Unsigned Word
ADDU	Add Unsigned Word
CLO	Count Leading Ones in Word
CLZ	Count Leading Zeros in Word
DADD	Doubleword Add
DADDI	Doubleword Add immediate
DADDIU	Doubleword Add Immediate Unsigned
DADDU	Doubleword Add Unsigned
DCLO	Count Leading Ones in Doubleword
DCLZ	Count Leading Zeros in Doubleword
DDIV	Doubleword Divide
DDIVU	Doubleword Divide Unsigned
DIV	Divide Word
DIVU	Divide Unsigned Word
DMULT	Doubleword Multiply

Table 3.1 CPU Arithmetic Instructions (Continued)

Mnemonic	Instruction	
DMULTU	Doubleword Multiply Unsigned	
DSUB	Doubleword Subtract	
DSUBU	Doubleword Subtract Unsigned	
MADD	Multiply and Add Word to Hi, Lo	
MADDU	Multiply and Add Unsigned Word to Hi, Lo	
MSUB	Multiply and Subtract Word to Hi, Lo	
MSUBU	Multiply and Subtract Unsigned Word to Hi, Lo	
MUL	Multiply Word to GPR	
MULT	Multiply Word	
MULTU	Multiply Unsigned Word	
SEB	Sign-Extend Byte	Release 2 & subsequent
SEH	Sign-Extend Halfword	Release 2 & subsequent
SLT	Set on Less Than	
SLTI	Set on Less Than Immediate	
SLTIU	Set on Less Than Immediate Unsigned	
SLTU	Set on Less Than Unsigned	
SUB	Subtract Word	
SUBU	Subtract Unsigned Word	

Table 3.2 CPU Branch and Jump Instructions

Mnemonic	Instruction
B	Unconditional Branch
BAL	Branch and Link
BEQ	Branch on Equal
BGEZ	Branch on Greater Than or Equal to Zero
BGEZAL	Branch on Greater Than or Equal to Zero and Link
BGTZ	Branch on Greater Than Zero
BLEZ	Branch on Less Than or Equal to Zero
BLTZ	Branch on Less Than Zero
BLTZAL	Branch on Less Than Zero and Link
BNE	Branch on Not Equal

Table 3.2 CPU Branch and Jump Instructions (Continued)

Mnemonic	Instruction	
J	Jump	
JAL	Jump and Link	
JALR	Jump and Link Register	
JALR.HB	Jump and Link Register with Hazard Barrier	Release 2 & subsequent
JALX	Jump and Link Exchange	microMIPS or MIPS16e also implemented
JR	Jump Register	
JR.HB	Jump Register with Hazard Barrier	Release 2 & subsequent

Table 3.3 CPU Instruction Control Instructions

Mnemonic	Instruction	
EHB	Execution Hazard Barrier	Release 2 & subsequent
NOP	No Operation	
PAUSE	Wait for LLBit to Clear	
SSNOP	Superscalar No Operation	Release 2.6 & subsequent

Table 3.4 CPU Load, Store, and Memory Control Instructions

Mnemonic	Instruction	
LB	Load Byte	
LBE	Load Byte EVA	Release 3.03 & subsequent
LBU	Load Byte Unsigned	
LBUE	Load Byte Unsigned EVA	Release 3.03 & subsequent
LD	Load Doubleword	
LDL	Load Doubleword LEft	
LDR	Load Doubleword Right	
LH	Load Halfword	
LHE	Load Halfword EVA	Release 3.03 & subsequent
LHU	Load Halfword Unsigned	

Table 3.4 CPU Load, Store, and Memory Control Instructions (Continued)

Mnemonic	Instruction	
LHUE	Load Halfword Unsigned EVA	Release 3.03 & subsequent
LL	Load Linked Word	
LLE	Load Linked Word-EVA	Release 3.03 & subsequent
LLD	Load Linked Doubleword	
LW	Load Word	
LWE	Load Word EVA	Release 3.03 & subsequent
LWL	Load Word Left	
LWLE	Load Word Left EVA	Release 3.03 & subsequent
LWR	Load Word Right	
LWRE	Load Word Right EVA	Release 3.03 & subsequent
LWU	Load Word Unsigned	
PREF	Prefetch	
PREFE	Prefetch-EVA	Release 3.03 & subsequent
SB	Store Byte	
SBE	Store Byte EVA	Release 3.03 & subsequent
SC	Store Conditional Word	
SCE	Store Conditional Word EVA	Release 3.03 & subsequent
SCD	Store Conditional Doubleword	
SD	Store Doubleword	
SDL	Store Doubleword LEft	
SDR	Store Doubleword Right	
SH	Store Halfword	
SHE	Store Halfword EVA	Release 3.03 & subsequent
SW	Store Word	
SWE	Store Word EVA	Release 3.03 & subsequent

Table 3.4 CPU Load, Store, and Memory Control Instructions (Continued)

Mnemonic	Instruction	
SWL	Store Word Left	Release 3.03 & subsequent
SWLE	Store Word Left EVA	
SWR	Store Word Right	
SWRE	Store Word Right EVA	Release 3.03 & subsequent
SYNC	Synchronize Shared Memory	
SYNCI	Synchronize Caches to Make Instruction Writes Effective	Release 2 & subsequent

Table 3.5 CPU Logical Instructions

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

Table 3.6 CPU Insert/Extract Instructions

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

Table 3.6 CPU Insert/Extract Instructions (Continued)

Mnemonic	Instruction	
INS	Insert Bit Field	Release 2 & subsequent
WSBH	Word Swap Bytes Within Halfwords	Release 2 & subsequent

Table 3.7 CPU Move Instructions

Mnemonic	Instruction	
MFHI	Move From HI Register	
MFLO	Move From LO Register	
MOVF	Move Conditional on Floating Point False	
MOVN	Move Conditional on Not Zero	
MOVT	Move Conditional on Floating Point True	
MOVZ	Move Conditional on Zero	
MTHI	Move To HI Register	
MTLO	Move To LO Register	
RDHWR	Read Hardware Register	Release 2 & subsequent

Table 3.8 CPU Shift Instructions

Mnemonic	Instruction	
DROTR	Doubleword Rotate Right	Release 2 & subsequent
DROTR32	Doubleword Rotate Right Plus 32	Release 2 & subsequent
DROTRV	Doubleword Rotate Right Variable	Release 2 & subsequent
DSLL	Doubleword Shift Left Logical	
DSLL32	Doubleword Shift Left Logical Plus 32	
DSLLV	Doubleword Shift Left Logical Variable	
DSRA	Doubleword Shift Right Arithmetic	
DSRA32	Doubleword Shift Right Arithmetic Plus 32	
DSRAV	Doubleword Shift Right Arithmetic Variable	
DSRL	Doubleword Shift Right Logical	
DSRL32	Doubleword Shift Right Logical Plus 32	
DSRLV	Doubleword Shift Right Logical Variable	
ROTR	Rotate Word Right	Release 2 & subsequent

Table 3.8 CPU Shift Instructions (Continued)

Mnemonic	Instruction	
ROTRV	Rotate Word Right Variable	Release 2 & subsequent
SLL	Shift Word Left Logical	
SLLV	Shift Word Left Logical Variable	
SRA	Shift Word Right Arithmetic	
SRAV	Shift Word Right Arithmetic Variable	
SRL	Shift Word Right Logical	
SRLV	Shift Word Right Logical Variable	

Table 3.9 CPU Trap Instructions

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

Table 3.10 Obsolete¹ CPU Branch Instructions

Mnemonic	Instruction
BEQL	Branch on Equal Likely
BGEZALL	Branch on Greater Than or Equal to Zero and Link Likely
BGEZL	Branch on Greater Than or Equal to Zero Likely

Table 3.10 Obsolete¹ CPU Branch Instructions (Continued)

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

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

Table 3.11 FPU Arithmetic Instructions

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

Table 3.12 FPU Branch Instructions

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

Table 3.13 FPU Compare Instructions

Mnemonic	Instruction
C.cond(fmt)	Floating Point Compare

Table 3.14 FPU Convert Instructions

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

Table 3.15 FPU Load, Store, and Memory Control Instructions

Mnemonic	Instruction	
LDC1	Load Doubleword to Floating Point	
LDXC1	Load Doubleword Indexed to Floating Point	64-bit FPU Only

Table 3.15 FPU Load, Store, and Memory Control Instructions (Continued)

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

Table 3.16 FPU Move Instructions

Mnemonic	Instruction	
CFC1	Move Control Word from Floating Point	
CTC1	Move Control Word to Floating Point	
DMFC1	Doubleword Move from Floating Point	
DMTC1	Doubleword Move to Floating Point	
MFC1	Move Word from Floating Point	
MFHC1	Move Word from High Half of Floating Point Register	Release 2 & subsequent
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	
MTHC1	Move Word to High Half of Floating Point Register	Release 2 & subsequent

Table 3.17 Obsolete¹ FPU Branch Instructions

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

The MIPS64® Instruction Set

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

Table 3.18 Coprocessor Branch Instructions

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

Table 3.19 Coprocessor Execute Instructions

Mnemonic	Instruction
COP2	Coprocessor Operation to Coprocessor 2

Table 3.20 Coprocessor Load and Store Instructions

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

Table 3.21 Coprocessor Move Instructions

Mnemonic	Instruction	
CFC2	Move Control Word from Coprocessor 2	
CTC2	Move Control Word to Coprocessor 2	
DMFC2	Doubleword Move from Coprocessor 2	
DMTC2	Doubleword Move to Coprocessor 2	
MFC2	Move Word from Coprocessor 2	
MFHC2	Move Word from High Half of Coprocessor 2 Register	Release 2 & subsequent
MTC2	Move Word to Coprocessor 2	
MTHC2	Move Word to High Half of Coprocessor 2 Register	Release 2 & subsequent

Table 3.22 Obsolete¹ Coprocessor Branch Instructions

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

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

Table 3.23 Privileged Instructions

Mnemonic	Instruction	
CACHE	Perform Cache Operation	
CACHEE	Perform Cache Operation EVA	Release 3.03 & subsequent
DI	Disable Interrupts	Release 2 & subsequent
DMFC0	Doubleword Move from Coprocessor 0	
DMTC0	Doubleword Move to Coprocessor 0	
EI	Enable Interrupts	Release 2 & subsequent
ERET	Exception Return	
MFC0	Move from Coprocessor 0	
MTC0	Move to Coprocessor 0	
RDPGPR	Read GPR from Previous Shadow Set	Release 2 & subsequent
TLBP	Probe TLB for Matching Entry	
TLBR	Read Indexed TLB Entry	
TLBWI	Write Indexed TLB Entry	
TLBWR	Write Random TLB Entry	
WAIT	Enter Standby Mode	
WRPGPR	Write GPR to Previous Shadow Set	Release 2 & subsequent

Table 3.24 EJTAG Instructions

Mnemonic	Instruction
DERET	Debug Exception Return
SDBBP	Software Debug Breakpoint

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

6 5 5 5 5 6

Format: ABS.fmt
 ABS.S fd, fs
 ABS.D fd, fs
 ABS.PS fd, fs

MIPS32
 MIPS32
MIPS64, MIPS32 Release 2

Purpose: Floating Point Absolute Value

Description: FPR[fd] $\leftarrow \text{abs}(\text{FPR}[fs])$

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

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

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

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

Restrictions:

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

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

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

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

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

Format: ADD rd, rs, rt

MIPS32

Purpose: Add Word

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

Description: $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

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

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

Restrictions:

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

Operation:

```

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

```

Exceptions:

Integer Overflow

Programming Notes:

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

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

Format: ADD fmt

ADD.S fd, fs, ft

MIPS32

ADD.D fd, fs, ft

MIPS32

ADD.PS fd, fs, ft

MIPS64, MIPS32 Release 2

Purpose: Floating Point Add

To add floating point values

Description: FPR[fd] \leftarrow FPR[fs] + FPR[ft]

The value in FPR *ft* is added to the value in FPR *fs*. The result is calculated to infinite precision, rounded by using to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. ADD.PS adds the upper and lower halves of FPR *fs* and FPR *ft* independently, and ORs together any generated exceptions.

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

Restrictions:

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

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

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

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow

31	26 25	21 20	16 15	0
6	5	5	16	
ADDI 001000	rs	rt	immediate	

Format: ADDI rt, rs, immediate

MIPS32

Purpose: Add Immediate Word

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

Description: $GPR[rt] \leftarrow GPR[rs] + \text{immediate}$

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 sign-extended and placed into GPR *rt*.

Restrictions:

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

Operation:

```

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

```

Exceptions:

Integer Overflow

Programming Notes:

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

31	26 25	21 20	16 15	0
ADDIU 001001	rs	rt	immediate	

Format: ADDIU rt, rs, immediate

MIPS32

Purpose: Add Immediate Unsigned Word

To add a constant to a 32-bit integer

Description: $GPR[rt] \leftarrow GPR[rs] + \text{immediate}$

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

No Integer Overflow exception occurs under any circumstances.

Restrictions:

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

Operation:

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

Exceptions:

None

Programming Notes:

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

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

Format: ADDU rd, rs, rt

MIPS32

Purpose: Add Unsigned Word

To add 32-bit integers

Description: $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

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

No Integer Overflow exception occurs under any circumstances.

Restrictions:

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

Operation:

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

Exceptions:

None

Programming Notes:

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

31	26 25	21 20	16 15	11 10	6 5	0
COP1X 010011	rs	ft	fs	fd	ALNV.PS 011110	
6	5	5	5	5	6	

Format: ALNV.PS fd, fs, ft, rs

MIPS64, MIPS32 Release 2

Purpose: Floating Point Align Variable

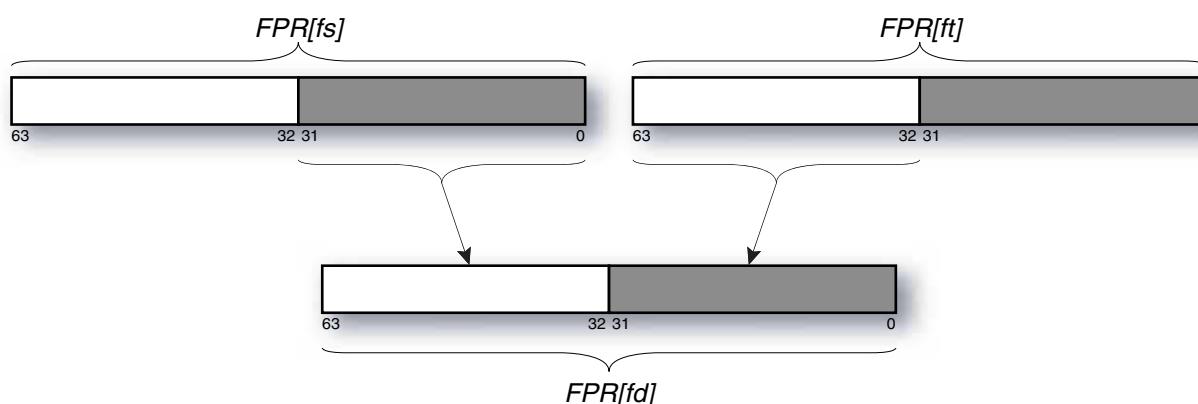
To align a misaligned pair of paired single values

Description: $FPR[fd] \leftarrow \text{ByteAlign}(GPR[rs]_{2..0}, FPR[fs], FPR[ft])$

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

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

Figure 3.1 Example of an ALNV.PS Operation



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

Restrictions:

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

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

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

Operation:

```

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

```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Programming Notes:

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

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

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

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

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

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

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

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

Format: AND rd, rs, rt

MIPS32

Purpose: And

To do a bitwise logical AND

Description: GPR[rd] \leftarrow GPR[rs] AND GPR[rt]

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

Restrictions:

None

Operation:

GPR[rd] \leftarrow GPR[rs] and GPR[rt]

Exceptions:

None

31	26 25	21 20	16 15	0
ANDI 001100	rs	rt	immediate	

Format: ANDI rt, rs, immediate

MIPS32

Purpose: And Immediate

To do a bitwise logical AND with a constant

Description: $GPR[rt] \leftarrow GPR[rs] \text{ AND immediate}$

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

Restrictions:

None

Operation:

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

Exceptions:

None

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

6 5 5 16

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

To do an unconditional branch

Description: branch

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

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

Restrictions:

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

Operation:

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

Exceptions:

None

Programming Notes:

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

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

6 5 5 16

Format: BAL offset**Assembly Idiom****Purpose:** Branch and Link

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 2bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

Restrictions:

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

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

Operation:

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

Exceptions:

None

Programming Notes:

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

31	26 25	21 20	18 17 16 15	0
COP1 010001	BC 010000	cc	nd 0	tf 0

6 5 3 1 16 offset

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 FPConditionCode(cc) = 0 then branch

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

Restrictions:

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

Operation:

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

```

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Programming Notes:

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

Historical Information:

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

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

In the MIPS I, II, and III architectures there must be at least one instruction between the compare instruction that sets

the condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.

31	26 25	21 20	18 17	16	15	0
COP1 010001	BC 010000	cc	nd 1	tf 0		offset

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 FPConditionCode(cc) = 0 then branch_likely

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

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

Restrictions:

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

Operation:

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

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Programming Notes:

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

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

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

encouraged to use the 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 010001	BC 01000	cc	nd 0	tf 1

6 5 3 1 16 offset

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 FPConditionCode(cc) = 1 then branch

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

Restrictions:

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

Operation:

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

```

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Programming Notes:

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

Historical Information:

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

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

In the MIPS I, II, and III architectures there must be at least one instruction between the compare instruction that sets

the condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.

31	26 25	21 20	18 17 16 15	0
COP1 010001	BC 010000	cc	nd 1	tf 1

6 5 3 1 16

offset

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 FPConditionCode(cc) = 1 then branch_likely

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

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

Restrictions:

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

Operation:

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

```

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Programming Notes:

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

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

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

will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BC1T instruction instead.

Historical Information:

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

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

In the MIPS II 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 010010	BC 01000	cc	nd 0	tf 0

6 5 3 1 16

offset

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 COP2Condition(cc) = 0 then branch

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

Restrictions:

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

Operation:

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

```

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Programming Notes:

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

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

6 5 3 1 16

offset

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 COP2Condition(cc) = 0 then branch_likely`
 An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by *cc* is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

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

Operation:

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

```

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Programming Notes:

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

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

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

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

6 5 3 1 16

offset

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 COP2Condition(cc) = 1 then branch

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

Restrictions:

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

Operation:

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

```

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Programming Notes:

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

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

6 5 3 1 16 offset

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 COP2Condition(cc) = 1 then branch_likely

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

Restrictions:

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

Operation:

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

```

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Programming Notes:

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

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

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

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

6 5 5 16

Format: BEQ rs, rt, offset**MIPS32****Purpose:** Branch on Equal

To compare GPRs then do a PC-relative conditional branch

Description: if $\text{GPR}[\text{rs}] = \text{GPR}[\text{rt}]$ then branchAn 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.If the contents of GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.**Restrictions:**Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.**Operation:**

```

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

Exceptions:

None

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

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

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

6 5 5 16

Format: BEQL rs, rt, offset**MIPS32****Purpose:** Branch on Equal Likely

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

Description: if $GPR[rs] = GPR[rt]$ then branch_likelyAn 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.If the contents of GPR *rs* and GPR *rt* are equal, branch to the target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.**Restrictions:**Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.**Operation:**

```

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

Exceptions:

None

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

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

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

Historical Information:

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

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

6 5 5 16

Format: BGEZ rs, offset**MIPS32****Purpose:** Branch on Greater Than or Equal to Zero

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

Description: if $\text{GPR}[\text{rs}] \geq 0$ then branchAn 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 ± 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 000001	rs	BGEZAL 10001		offset

6 5 5 16

Format: BGEZAL rs, offset**MIPS32****Purpose:** Branch on Greater Than or Equal to Zero and Link

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

Description: if $\text{GPR}[\text{rs}] \geq 0$ 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 ± 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:** Branch on Greater Than or Equal to Zero and Link Likely

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

Description: if $\text{GPR}[\text{rs}] \geq 0$ 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 ± 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 000001	rs	BGEZL 00011		offset

6 5 5 16

Format: BGEZL rs, offset**MIPS32****Purpose:** Branch on Greater Than or Equal to Zero Likely

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

Description: if $\text{GPR}[\text{rs}] \geq 0$ then branch_likelyAn 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 ± 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.

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

6 5 5 16

Format: BGTZ rs, offset**MIPS32****Purpose:** Branch on Greater Than Zero

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

Description: if GPR[rs] > 0 then branchAn 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 ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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

6 5 5 16

Format: BGTZL rs, offset**MIPS32****Purpose:** Branch on Greater Than Zero Likely

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

Description: if GPR[rs] > 0 then branch_likelyAn 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 ± 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.

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

6 5 5 16

Format: BLEZ rs, offset**MIPS32****Purpose:** Branch on Less Than or Equal to Zero

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

Description: if $\text{GPR}[\text{rs}] \leq 0$ then branchAn 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 ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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

6 5 5 16

Format: BLEZL rs, offset**MIPS32****Purpose:** Branch on Less Than or Equal to Zero Likely

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

Description: if $\text{GPR}[\text{rs}] \leq 0$ then branch_likelyAn 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 ± 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.

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

6 5 5 16

Format: BLTZ rs, offset**MIPS32****Purpose:** Branch on Less Than Zero

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

Description: if GPR[rs] < 0 then branchAn 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 ± 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:** Branch on Less Than Zero and Link

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

Description: if GPR[rs] < 0 then procedure_call

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

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.**Restrictions:**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 ± 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:** Branch on Less Than Zero and Link Likely

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

Description: if GPR[rs] < 0 then procedure_call_likely

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

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.**Restrictions:**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 ± 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 000001	rs	BLTZL 00010	offset	

6 5 5 16

Format: BLTZL rs, offset**MIPS32****Purpose:** Branch on Less Than Zero Likely

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

Description: if GPR[rs] < 0 then branch_likelyAn 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 ± 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.

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

Format: BNE rs, rt, offset**MIPS32****Purpose:** Branch on Not Equal

To compare GPRs then do a PC-relative conditional branch

Description: if $GPR[rs] \neq GPR[rt]$ then branchAn 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.If the contents of GPR *rs* and GPR *rt* are not equal, branch to the effective target address after the instruction in the delay slot is executed.**Restrictions:**Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.**Operation:**

```

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

Exceptions:

None

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

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

6 5 5 16

Format: BNEL rs, rt, offset**MIPS32****Purpose:** Branch on Not Equal Likely

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

Description: if GPR[rs] ≠ GPR[rt] then branch_likelyAn 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.If the contents of GPR *rs* and GPR *rt* are not equal, branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.**Restrictions:**Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.**Operation:**

```

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

Exceptions:

None

Programming Notes:

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

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

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

Historical Information:

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

31	26 25		6 5	0
SPECIAL 000000		code	BREAK 001101	

6 20 6

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

To cause a Breakpoint exception

Description:

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

Restrictions:

None

Operation:

SignalException(Breakpoint)

Exceptions:

Breakpoint

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

6 5 5 5 3 1 1 2 4

Format: C.cond(fmt)

C.cond.S fs, ft (cc = 0 implied) MIPS32

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

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

C.cond.S cc, fs, ft MIPS32

C.cond.D cc, fs, ft MIPS32

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

Purpose: Floating Point Compare

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

Description: `FPConditionCode(cc) ← FPR[fs] compare_cond FPR[ft]`

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

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

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

If one of the values is an SNaN, or *cond*₃ is set and at least one of the values is a QNaN, an Invalid Operation condition is raised and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written and an Invalid Operation exception is taken immediately. Otherwise, the Boolean result is written into condition code *CC*.

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

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

Logical negation of a compare result allows eight distinct comparisons to test for the 16 predicates as shown in [Table 3.25](#). Each mnemonic tests for both a predicate and its logical negation. For each mnemonic, *compare* tests the truth of the first predicate. When the first predicate is true, the result is true as shown in the “If Predicate Is True” column, and the second predicate must be false, and vice versa. (Note that the False predicate is never true and False/True do not follow the normal pattern.)

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

[Table 3.26](#) shows another set of eight compare operations, distinguished by a *cond*₃ value of 1 and testing the same 16 conditions. For these additional comparisons, if at least one of the operands is a NaN, including Quiet NaN, then an Invalid Operation condition is raised. If the Invalid Operation condition is enabled in the *FCSR*, an Invalid Operation

exception occurs.

Table 3.25 FPU Comparisons Without Special Operand Exceptions

Instruction Cond Mnemonic	Comparison Predicate Name of Predicate and Logically Negated Predicate (Abbreviation)	Comparison CC Result				Instruction Condition Field 3 2..0	
		Relation Values					
		>	<	=	?		
F	False [this predicate is always False]	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		
EQ	Equal	F	F	T	F	T	2
	Not Equal (NEQ)	T	T	F	T		
UEQ	Unordered or Equal	F	F	T	T	T	3
	Ordered or Greater Than or Less Than (OGL)	T	T	F	F		
OLT	Ordered or Less Than	F	T	F	F	T	4
	Unordered or Greater Than or Equal (UGE)	T	F	T	T		
ULT	Unordered or Less Than	F	T	F	T	T	5
	Ordered or Greater Than or Equal (OGE)	T	F	T	F		
OLE	Ordered or Less Than or Equal	F	T	T	F	T	6
	Unordered or Greater Than (UGT)	T	F	F	T		
ULE	Unordered or Less Than or Equal	F	T	T	T	T	7
	Ordered or Greater Than (OGT)	T	F	F	F		

Key: ? = *unordered*, > = *greater than*, < = *less than*, = is *equal*, T = *True*, F = *False*

Table 3.26 FPU Comparisons With Special Operand Exceptions for QNaNs

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

Key: ? = *unordered*, > = *greater than*, < = *less than*, = is *equal*, T = *True*, F = *False*

Restrictions:

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

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

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

Operation:

```

if SNaN(ValueFPR(fs, fmt)) or SNaN(ValueFPR(ft, fmt)) or
   QNaN(ValueFPR(fs, fmt)) or QNaN(ValueFPR(ft, fmt)) then
   less ← false
   equal ← false
   unordered ← true
   if (SNaN(ValueFPR(fs, fmt)) or SNaN(ValueFPR(ft, fmt))) or
      (cond3 and (QNaN(ValueFPR(fs, fmt)) or QNaN(ValueFPR(ft, fmt)))) then
         SignalException(InvalidOperation)
      endif
   else
      less ← ValueFPR(fs, fmt) <fmt ValueFPR(ft, fmt)
      equal ← ValueFPR(fs, fmt) =fmt ValueFPR(ft, fmt)
      unordered ← false
   endif
endif

```

```

condition ← (cond2 and less) or (cond1 and equal)
      or (cond0 and unordered)
SetFPCConditionCode(cc, condition)

```

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

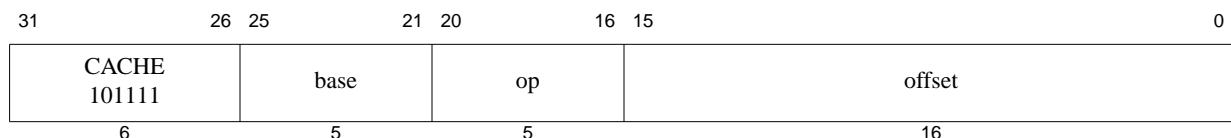
Programming Notes:

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

```

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

```



Format: CACHE op, offset(base)

MIPS32

Purpose: Perform Cache Operation

To perform the cache operation specified by op.

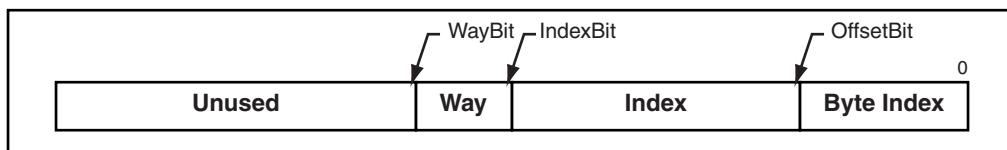
Description:

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

Table 3.27 Usage of Effective Address

Operation Requires an	Type of Cache	Usage of Effective Address
Address	Virtual	The effective address is used to address the cache. An address translation may or may not be performed on the effective address (with the possibility that a TLB Refill or TLB Invalid exception might occur)
Address	Physical	The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache
Index	N/A	<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. As such, an unmapped address (such as within kseg0) should always be used for cache operations that require an index. See the Programming Notes section below.</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> <pre> OffsetBit ← Log2(BPT) IndexBit ← Log2(CS / A) WayBit ← IndexBit + Ceiling(Log2(A)) Way ← Addr_{WayBit-1..IndexBit} Index ← Addr_{IndexBit-1..OffsetBit} </pre> <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 3.2 Usage of Address Fields to Select Index and Way



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

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

A Cache Error exception may occur as a by-product of some operations performed by this instruction. For example, if a Writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error. However, cache error exceptions must not be triggered by an Index Load Tag or Index Store tag operation, as these operations are used for initialization and diagnostic purposes.

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

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

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

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

Table 3.28 Encoding of Bits[17:16] of CACHE Instruction

Code	Name	Cache
0b00	I	Primary Instruction
0b01	D	Primary Data or Unified Primary
0b10	T	Tertiary
0b11	S	Secondary

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

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

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

memory hierarchy.

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

Table 3.29 Encoding of Bits [20:18] of the CACHE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b000	I	Index Invalidate	Index	<p>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.</p>	Required
	D	Index Writeback Invalidate / Index Invalidate	Index	<p>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.</p>	Required
	S, T	Index Writeback Invalidate / Index Invalidate	Index	<p>For a write-through cache: Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. Note that Index Store Tag should be used to initialize the cache at power up.</p>	Required if S, T cache is implemented
0b001	All	Index Load Tag	Index	<p>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.</p>	Recommended

Table 3.29 Encoding of Bits [20:18] of the CACHE Instruction (Continued)

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b010	All	Index Store Tag	Index	Write the tag for the cache block at the specified index from the <i>TagLo</i> and <i>TagHi</i> Coprocessor 0 registers. This operation must not cause a Cache Error Exception. This required encoding may be used by software to initialize the entire instruction or data caches by stepping through all valid indices. Doing so requires that the <i>TagLo</i> and <i>TagHi</i> registers associated with the cache be initialized first.	Required
0b011	All	Implementation Dependent	Unspecified	Available for implementation-dependent operation.	Optional
0b100	I, D	Hit Invalidate	Address	If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache.	Required (Instruction Cache Encoding Only), Recommended otherwise
	S, T	Hit Invalidate	Address	In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Optional, if Hit_Invalidate_D is implemented, the S and T variants are recommended.
0b101	I	Fill	Address	Fill the cache from the specified address.	Recommended
	D	Hit Writeback Invalidate / Hit Invalidate	Address	For a write-back cache: If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache.	Required
	S, T	Hit Writeback Invalidate / Hit Invalidate	Address	In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Required if S, T cache is implemented

Table 3.29 Encoding of Bits [20:18] of the CACHE Instruction (Continued)

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b110	D	Hit Writeback	Address	If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state. For a write-through cache, this operation may be treated as a nop.	Recommended
	S, T	Hit Writeback	Address	In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Optional, if Hit_Writeback_D is implemented, the S and T variants are recommended.
0b111	I, D	Fetch and Lock	Address	If the cache does not contain the specified address, fill it from memory, performing a writeback if required, and set the state to valid and locked. If the cache already contains the specified address, set the state to locked. In set-associative or fully-associative caches, the way selected on a fill from memory is implementation dependent. The lock state may be cleared by executing an Index Invalidate, Index Writeback Invalidate, Hit Invalidate, or Hit Writeback Invalidate operation to the locked line, or via an Index Store Tag operation to the line that clears the lock bit. Note that clearing the lock state via Index Store Tag is dependent on the implementation-dependent cache tag and cache line organization, and that Index and Index Writeback Invalidate operations are dependent on cache line organization. Only Hit and Hit Writeback Invalidate operations are generally portable across implementations. It is implementation dependent whether a locked line is displaced as the result of an external invalidate or intervention that hits on the locked line. Software must not depend on the locked line remaining in the cache if an external invalidate or intervention would invalidate the line if it were not locked. It is implementation dependent whether a Fetch and Lock operation affects more than one line. For example, more than one line around the referenced address may be fetched and locked. It is recommended that only the single line containing the referenced address be affected.	Recommended

Restrictions:

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

The operation of this instruction is **UNDEFINED** if the operation requires an address, and that address is 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 this instruction is used to lock all ways of a cache at a specific cache index, the behavior of that cache to subsequent cache misses to that cache index is **UNDEFINED**.

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

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

Operation:

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

Exceptions:

TLB Refill Exception.

TLB Invalid Exception

Coprocessor Unusable Exception

Address Error Exception

Cache Error Exception

Bus Error Exception

Programming Notes:

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

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

31	26 25	21 20	16 15	7 6 5	0
SPECIAL3 011111	base	op	offset	0	CACHEE 011011

6 5 5 9 1 6

Format: CACHEE op, offset(base)

MIPS32

Purpose: Perform Cache Operation EVA

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

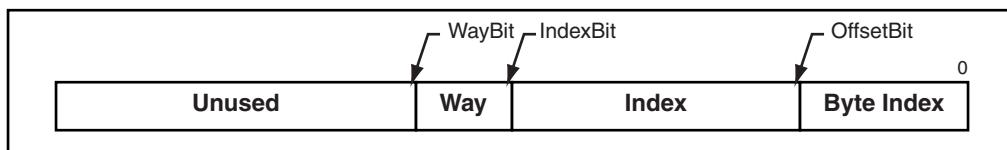
Description:

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

Table 3.30 Usage of Effective Address

Operation Requires an	Type of Cache	Usage of Effective Address
Address	Virtual	The effective address is used to address the cache. An address translation may or may not be performed on the effective address (with the possibility that a TLB Refill or TLB Invalid exception might occur)
Address	Physical	The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache
Index	N/A	<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. As such, a kseg0 address should always be used for cache operations that require an index. See the Programming Notes section below.</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> <pre> OffsetBit ← Log2(BPT) IndexBit ← Log2(CS / A) WayBit ← IndexBit + Ceiling(Log2(A)) Way ← Addr_{WayBit-1..IndexBit} Index ← Addr_{IndexBit-1..OffsetBit} </pre> <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 3.3 Usage of Address Fields to Select Index and Way



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

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

A Cache Error exception may occur as a by-product of some operations performed by this instruction. For example, if a Writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error. However, cache error exceptions must not be triggered by an Index Load Tag or Index Store tag operation, as these operations are used for initialization and diagnostic purposes.

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

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

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

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

Table 3.31 Encoding of Bits[17:16] of CACHEE Instruction

Code	Name	Cache
0b00	I	Primary Instruction
0b01	D	Primary Data or Unified Primary
0b10	T	Tertiary
0b11	S	Secondary

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

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

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

of memory hierarchy.

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

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

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

Table 3.32 Encoding of Bits [20:18] of the CACHEE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b000	I	Index Invalidate	Index	<p>Set the state of the cache block at the specified index to invalid.</p> <p>This required encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.</p>	Required
	D	Index Writeback Invalidate / Index Invalidate	Index	<p>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.</p>	Required
	S, T	Index Writeback Invalidate / Index Invalidate	Index	<p>For a write-through cache: Set the state of the cache block at the specified index to invalid.</p> <p>This required encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. Note that Index Store Tag should be used to initialize the cache at power up.</p>	Required if S, T cache is implemented

Table 3.32 Encoding of Bits [20:18] of the CACHEE Instruction (Continued)

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b001	All	Index Load Tag	Index	Read the tag for the cache block at the specified index into the <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
0b010	All	Index Store Tag	Index	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. 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.	Required
0b011	All	Implementation Dependent	Unspecified	Available for implementation-dependent operation.	Optional
0b100	I, D	Hit Invalidate	Address	If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache.	Required (Instruction Cache Encoding Only), Recommended otherwise
	S, T	Hit Invalidate	Address	In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Optional, if Hit_Invalidate_D is implemented, the S and T variants are recommended.

Table 3.32 Encoding of Bits [20:18] of the CACHEE Instruction (Continued)

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b101	I	Fill	Address	Fill the cache from the specified address.	Recommended
	D	Hit Writeback Invalidate / Hit Invalidate	Address	For a write-back cache: If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache.	Required
	S, T	Hit Writeback Invalidate / Hit Invalidate	Address	In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Required if S, T cache is implemented
0b110	D	Hit Writeback	Address	If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state. For a write-through cache, this operation may be treated as a nop.	Recommended
	S, T	Hit Writeback	Address	In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Optional, if Hit_Writeback_D is implemented, the S and T variants are recommended.

Table 3.32 Encoding of Bits [20:18] of the CACHEE Instruction (Continued)

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b111	I, D	Fetch and Lock	Address	<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 CACHEE instruction is the target of an invalidate or a writeback invalidate.

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

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

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

Operation:

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

Exceptions:

TLB Refill Exception.
TLB Invalid Exception
Coprocessor Unusable Exception
Reserved Instruction
Address Error Exception
Cache Error Exception
Bus Error Exception

Programming Notes:

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

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

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

Format: CEIL.L fmtCEIL.L.S fd, fs
CEIL.L.D fd, fsMIPS64, MIPS32 Release 2
MIPS64, MIPS32 Release 2**Purpose:** Fixed Point Ceiling Convert to Long Fixed Point

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

Description: FPR[fd] \leftarrow convert_and_round(FPR[fs])The value in FPR *fs*, in format *fmt*, is converted to a value in 64-bit long fixed point format and rounding toward $+\infty$ (rounding mode 2). The result is placed in FPR *fd*.When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}-1$, is written to *fd*.**Restrictions:**The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed point; if they are not valid, the result is **UNPREDICTABLE**.The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.**Operation:**

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

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

6 5 5 5 5 6

Format: CEIL.W(fmt)CEIL.W.S fd, fs
CEIL.W.D fd, fsMIPS32
MIPS32**Purpose:** Floating Point Ceiling Convert to Word Fixed Point

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

Description: FPR[fd] \leftarrow convert_and_round(FPR[fs])The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format and rounding toward $+\infty$ (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

31	26 25	21 20	16 15	11 10	0
COP1 010001	CF 000010	rt	fs	0 000 0000 0000	11

Format: CFC1 rt, fs**MIPS32****Purpose:** Move Control Word From Floating Point

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

Description: GPR[rt] \leftarrow FP_Control[fs]Copy the 32-bit word from FP (coprocessor 1) control register *fs* into GPR *rt*, sign-extending it to 64 bits.**Restrictions:**There are a few control registers defined for the floating point unit. The result is **UNPREDICTABLE** if *fs* specifies a register that does not exist.**Operation:**

```

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

```

Exceptions:

Coprocessor Unusable, Reserved Instruction

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

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

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

6 5 5 16

Format: CFC2 rt, Impl**MIPS32**

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

Purpose: Move Control Word From Coprocessor 2

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

Description: GPR[rt] \leftarrow CP2CCR[Impl]

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

Restrictions:

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

Operation:

```
temp  $\leftarrow$  CP2CCR[Impl]
GPR[rt]  $\leftarrow$  sign_extend(temp)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

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

Format: CLO rd, rs**MIPS32****Purpose:** Count Leading Ones in Word

To count the number of leading ones in a word

Description: GPR[rd] \leftarrow count_leading_ones GPR[rs]Bits 31..0 of GPR *rs* are scanned from most significant to least significant bit. The number of leading ones is counted and the result is written to GPR *rd*. If all of bits **31..0** were set in GPR *rs*, the result written to GPR *rd* is 32.**Restrictions:**To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the *rt* and *rd* fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values.If GPR *rs* does not contain a sign-extended 32-bit value (bits **63..31** equal), then the results of the operation are **UNPREDICTABLE**.**Operation:**

```

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

```

Exceptions:

None

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

Format: COP2 func**MIPS32****Purpose:** Coprocessor Operation to Coprocessor 2

To perform an operation to Coprocessor 2

Description: CoprocessorOperation(2, cofun)

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

Restrictions:**Operation:**

CoprocessorOperation(2, cofun)

Exceptions:

Coprocessor Unusable

Reserved Instruction

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

Format: CLZ rd, rs**MIPS32****Purpose:** Count Leading Zeros in Word

Count the number of leading zeros in a word

Description: GPR[rd] \leftarrow count_leading_zeros GPR[rs]

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

Restrictions:To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the rt and rd fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the rt and rd fields of the instruction contain different values.If GPR rs does not contain a sign-extended 32-bit value (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.**Operation:**

```

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

```

Exceptions:

None

31	26 25	21 20	16 15	11 10	0
COP1 010001	CT 00110	rt	fs	0 000 0000 0000	11

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

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

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

Writing to the floating point *Control/Status* register, the *FCSR*, causes the appropriate exception if any *Cause* bit and its corresponding *Enable* bit are both set. The register is written before the exception occurs. Writing to *FEXR* to set a cause bit whose enable bit is already set, or writing to *FENR* to set an enable bit whose cause bit is already set causes the appropriate exception. The register is written before the exception occurs and the *EPC* register contains the address of the CTC1 instruction.

Restrictions:

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

Operation:

```

temp  $\leftarrow$  GPR[rt]31..0
if fs = 25 then /* FCCR */
    if temp31..8  $\neq$  024 then
        UNPREDICTABLE
    else
        FCSR  $\leftarrow$  temp7..1 || FCSR24 || temp0 || FCSR22..0
    endif
elseif fs = 26 then /* FEXR */
    if temp31..18  $\neq$  0 or temp11..7  $\neq$  0 or temp2..0  $\neq$  0 then
        UNPREDICTABLE
    else
        FCSR  $\leftarrow$  FCSR31..18 || temp17..12 || FCSR11..7 ||
            temp6..2 || FCSR1..0
    endif
elseif fs = 28 then /* FENR */
    if temp31..12  $\neq$  0 or temp6..3  $\neq$  0 then
        UNPREDICTABLE
    else
        FCSR  $\leftarrow$  FCSR31..25 || temp2 || FCSR23..12 || temp11..7
            || FCSR6..2 || temp1..0
    endif
elseif fs = 31 then /* FCSR */
    if (FCSRImpl field is not implemented) and (temp22..18  $\neq$  0) then
        UNPREDICTABLE
    elseif (FCSRImpl field is implemented) and temp20..18  $\neq$  0 then
        UNPREDICTABLE
    else
        FCSR  $\leftarrow$  temp
    endif
else

```

```
UNPREDICTABLE
endif
CheckFPEException()
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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

Historical Information:

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

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

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

6 5 5 16

Format: CTC2 rt, Impl**MIPS32**

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

Purpose: Move Control Word to Coprocessor 2

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

Description: CP2CCR[Impl] \leftarrow GPR[rt]

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

Restrictions:

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

Operation:

```
temp  $\leftarrow$  GPR[rt]31..0
CP2CCR[Impl]  $\leftarrow$  temp
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

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

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

MIPS32
 MIPS32
MIPS64, MIPS32 Release 2

Purpose: Floating Point Convert to Double Floating Point

To convert an FP or fixed point value to double FP

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

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

Restrictions:

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

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

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

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	0 00000	fs	fd	CVT.L 100101	

6 5 5 5 5 6

Format: CVT.L fmt
 CVT.L.S fd, fs
 CVT.L.D fd, fs

MIPS64, MIPS32 Release 2
MIPS64, MIPS32 Release 2

Purpose: Floating Point Convert to Long Fixed Point

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

Description: FPR[fd] \leftarrow convert_and_round(FPR[fs])

Convert the value in format *fmt* in FPR *fs* to long fixed point format and round according to the current rounding mode in *FCSR*. The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}-1$, is written to *fd*.

Restrictions:

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

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

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

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact,

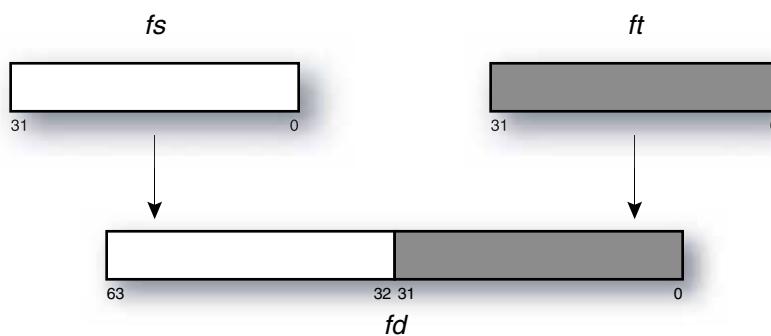
31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt 10000	ft	fs	fd	CVT.PS 100110	
6	5	5	5	5	6	

Format: CVT.PS.S fd, fs, ft**MIPS64, MIPS32 Release 2****Purpose:** Floating Point Convert Pair to Paired Single

To convert two FP values to a paired single value

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

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



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

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

Restrictions:The fields fs and ft must specify FPRs valid for operands of type S; if they are not valid, the result is **UNPREDICTABLE**.The operand must be a value in format S; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.**Operation:**

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation

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

Format: CVT.S fmt

CVT.S.D fd, fs

MIPS32

CVT.S.W fd, fs

MIPS32

CVT.S.L fd, fs

MIPS64, MIPS32 Release 2

Purpose: Floating Point Convert to Single Floating Point

To convert an FP or fixed point value to single FP

Description: FPR[fd] \leftarrow convert_and_round(FPR[fs])The value in FPR *fs*, in format *fmt*, is converted to a value in single floating point format and rounded according to the current rounding mode in FCSR. The result is placed in FPR *fd*.**Restrictions:**The fields *fs* and *fd* must specify valid FPRs—*fs* for type *fmt* and *fd* for single floating point. If they are not valid, the result is **UNPREDICTABLE**.The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.For CVT.S.L, the result of this instruction is **UNPREDICTABLE** if the processor is executing in 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 10110	0 00000	fs	fd	CVT.S.PL 101000	

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

Floating Point Convert Pair Lower to Single Floating Point

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

Description: FPR[fd] \leftarrow FPR[fs]_{31..0}The lower paired single value in FPR *fs*, in format *PS*, is converted to a value in single floating point format. The result is placed in FPR *fd*. This instruction can be used to isolate the lower half of a paired single value.

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

Restrictions:The fields *fs* and *fd* must specify valid FPRs—*fs* for type *PS* and *fd* for single floating point. If they are not valid, the result is **UNPREDICTABLE**.The operand must be a value in format *PS*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.The result of CVT.S.PL is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.**Operation:**

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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

Format: CVT.S.PU fd, fs**MIPS64, MIPS32 Release 2****Purpose:** Floating Point Convert Pair Upper to Single Floating Point

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

Description: FPR[fd] \leftarrow FPR[fs]_{63..32}The upper paired single value in FPR *fs*, in format *PS*, is converted to a value in single floating point format. The result is placed in FPR *fd*. This instruction can be used to isolate the upper half of a paired single value.

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

Restrictions:The fields *fs* and *fd* must specify valid FPRs—*fs* for type *PS* and *fd* for single floating point. If they are not valid, the result is **UNPREDICTABLE**.The operand must be a value in format *PS*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.The result of CVT.S.PU is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.**Operation:**

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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(fmt)
 CVT.W.S fd, fs
 CVT.W.D fd, fs

MIPS32
 MIPS32

Purpose: Floating Point Convert to Word Fixed Point

To convert an FP value to 32-bit fixed point

Description: FPR[fd] \leftarrow convert_and_round(FPR[fs])

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

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

Restrictions:

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

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

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

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

6 5 5 5 5 6

Format: DADD rd, rs, rt**MIPS64****Purpose:** Doubleword Add

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

Description: $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

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

Restrictions:**Operation:**

```

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

```

Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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

31	26 25	21 20	16 15	0
DADDI 011000	rs	rt	immediate	

6 5 5 16

Format: DADDI rt, rs, immediate**MIPS64****Purpose:** Doubleword Add Immediate

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

Description: GPR[rt] \leftarrow GPR[rs] + immediate

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

Restrictions:**Operation:**

```

temp  $\leftarrow$  (GPR[rs]63 || GPR[rs]) + sign_extend(immediate)
if (temp64  $\neq$  temp63) then
    SignalException(IntegerOverflow)
else
    GPR[rt]  $\leftarrow$  temp63..0
endif

```

Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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

31	26 25	21 20	16 15	0
DADDIU 011001	rs	rt	immediate	

Format: DADDIU rt, rs, immediate

MIPS64

Purpose: Doubleword Add Immediate Unsigned

To add a constant to a 64-bit integer

Description: $\text{GPR}[rt] \leftarrow \text{GPR}[rs] + \text{immediate}$

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

No Integer Overflow exception occurs under any circumstances.

Restrictions:

Operation:

$\text{GPR}[rt] \leftarrow \text{GPR}[rs] + \text{sign_extend}(\text{immediate})$

Exceptions:

Reserved Instruction

Programming Notes:

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

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	DADDU 101101	6

6 5 5 5 5 6

Format: DADDU rd, rs, rt**MIPS64****Purpose:** Doubleword Add Unsigned

To add 64-bit integers

Description: $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$ The 64-bit doubleword value in GPR *rt* is added to the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:**Operation:**
$$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$$
Exceptions:

Reserved Instruction

Programming Notes:

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

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL2 011100	rs	rt	rd	0 00000	DCLO 100101	6

Format: DCLO rd, rs

MIPS64

Purpose: Count Leading Ones in Doubleword

To count the number of leading ones in a doubleword

Description: GPR[rd] \leftarrow count_leading_ones GPR[rs]

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

Restrictions:

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

Operation:

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

Exceptions:

None

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

Format: DCLZ rd, rs

MIPS64

Purpose: Count Leading Zeros in Doubleword

To count the number of leading zeros in a doubleword

Description: GPR[rd] \leftarrow count_leading_zeros GPR[rs]

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

Restrictions:

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

Operation:

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

Exceptions:

None

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

6 5 5 10 6

Format: DDIV rs, rt**MIPS64****Purpose:** Doubleword Divide

To divide 64-bit signed integers

Description: $(LO, HI) \leftarrow GPR[rs] / GPR[rt]$

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

No arithmetic exception occurs under any circumstances.

Restrictions:

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

Operation:

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

Exceptions:

Reserved Instruction

Programming Notes:

See “Programming Notes” for the [DIV](#) instruction.

Historical Perspective:

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

31	26 25	21 20	16 15	6 5	0
SPECIAL 000000	rs	rt	0 00 0000 0000	DDIVU 011111	6

6 5 5 10 6

Format: DDIVU rs, rt**MIPS64****Purpose:** Doubleword Divide Unsigned

To divide 64-bit unsigned integers

Description: $(LO, HI) \leftarrow GPR[rs] / GPR[rt]$

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

No arithmetic exception occurs under any circumstances.

Restrictions:

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

Operation:

```

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

```

Exceptions:

Reserved Instruction

Programming Notes:

See “Programming Notes” for the [DIV](#) instruction.

Historical Perspective:

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

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

6 1 19 6

Format: DERET**EJTAG****Purpose:** Debug Exception Return

To Return from a debug exception.

Description:

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

Restrictions:

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

If the *DEPC* register with the return address for the DERET was modified by an MTC0 or a DMTC0 instruction, a CP0 hazard exists that must be removed via software insertion of the appropriate number of SSNOP instructions (for implementations of Release 1 of the Architecture) or by an EHB, or other execution hazard clearing instruction (for implementations of Release 2 of the Architecture).

DERET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNC1 instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the DERET returns.

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

Operation:

```

DebugDM ← 0
DebugIEXI ← 0
if IsMIPS16Implemented() || (Config3ISA > 0) then
    PC ← DEPC63..1 || 0
    ISAMode ← DEPC0
else
    PC ← DEPC
endif
ClearHazards()

```

Exceptions:

Coprocessor Unusable Exception
 Reserved Instruction Exception

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL3 011111	rs	rt	msbd (size-1)	lsb (pos)	DEXT 000011	

6 5 5 5 5 6

Format: DEXT rt, rs, pos, size

MIPS64 Release 2

Purpose: Doubleword Extract Bit Field

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

Description: $\text{GPR}[rt] \leftarrow \text{ExtractField}(\text{GPR}[rs], \text{msbd}, \text{lsb})$

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

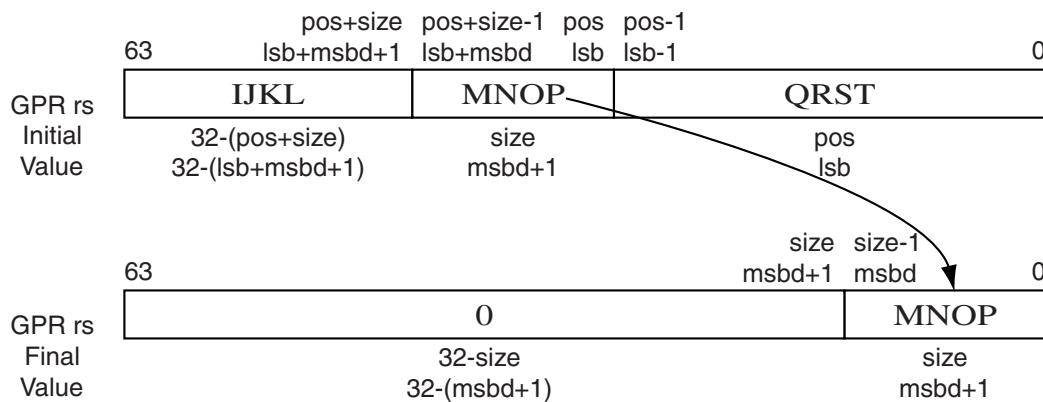
```
msbd ← size-1
lsb ← pos
msb ← lsb+msbd
```

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

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

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

Figure 3.4 Operation of the DEXT Instruction



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

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

Restrictions:

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

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

Operation:
$$\text{GPR}[rt] \leftarrow 0^{63-(msbd+1)} \mid\mid \text{GPR}[rs]_{msbd+lsb..lsb}$$
Exceptions:

Reserved Instruction

Programming Notes

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

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL3 011111	rs	rt	msbdminus32 (size-1-32)	lsb (pos)	DEXTM 000001	

6 5 5 5 5 6

Format: DEXTM rt, rs, pos, size**MIPS64 Release 2****Purpose:** Doubleword Extract Bit Field Middle

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

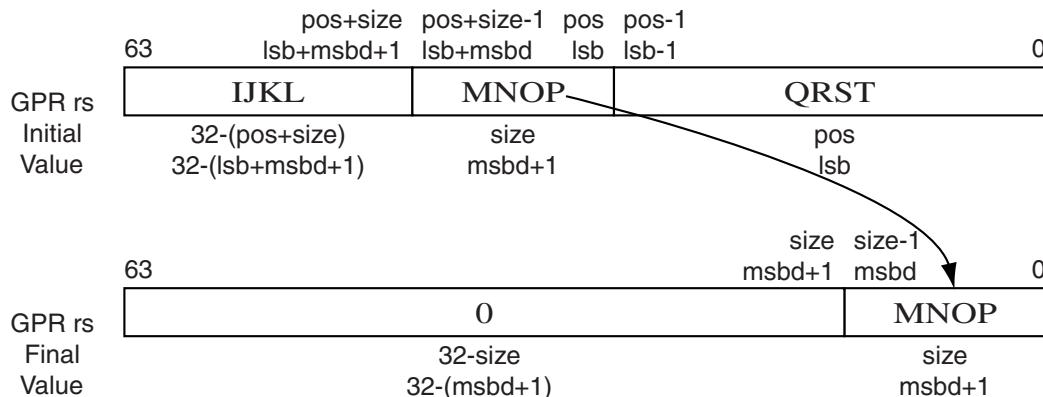
Description: $\text{GPR}[rt] \leftarrow \text{ExtractField}(\text{GPR}[rs], \text{msbd}, \text{lsb})$

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

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

For this instruction, the values of *pos* and *size* must satisfy all of the following relations:
$$\begin{aligned} 0 &\leq pos < 32 \\ 32 &< size \leq 64 \\ 32 &< pos+size \leq 64 \end{aligned}$$

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

Figure 3.5 Operation of the DEXTM Instruction

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

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

msbd	lsb	msb	pos	size	Instruction	Comment
$32 \leq msbd < 64$	$0 \leq lsb < 32$	$32 \leq msb < 64$	$0 \leq pos < 32$	$32 < size \leq 64$	DEXTM	The field is larger than 32 bits and starts in the right-most word of the doubleword

Restrictions:

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

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

Operation:

```

msbd ← msbdminus32 + 32
if ((lsb + msbd + 1) > 64) then
    UNPREDICTABLE
endif
GPR[rt] ← 063-(msbd+1) || GPR[rs]msbd+lsb..pos

```

Exceptions:

Reserved Instruction

Programming Notes

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

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL3 01111	rs	rt	msbd (size-1)	lsbminus32 (pos-32)	DEXTU 000010	

Format: DEXTU rt, rs, pos, size

MIPS64 Release 2

Purpose: Doubleword Extract Bit Field Upper

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

Description: $\text{GPR}[rt] \leftarrow \text{ExtractField}(\text{GPR}[rs], \text{msbd}, \text{lsb})$

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

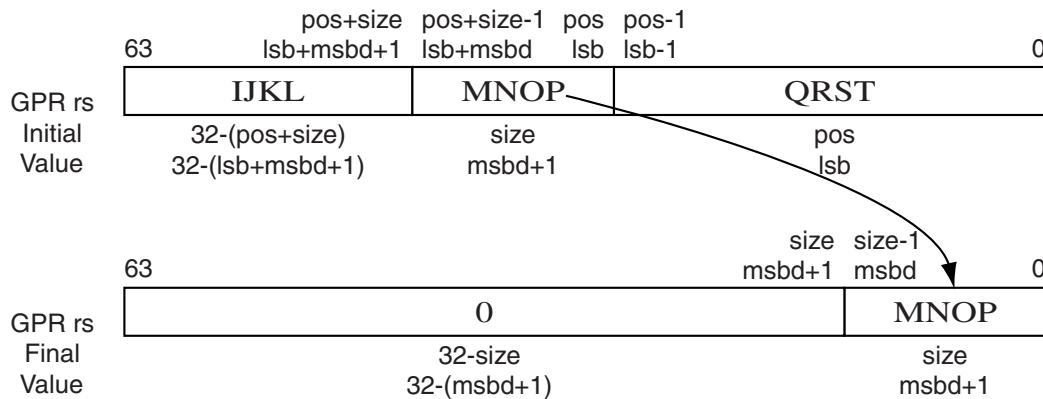
```
msbd ← size-1
lsbminus32 ← pos-32
lsb ← lsbminus32 + 32
msb ← lsb+msbd
```

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

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

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

Figure 3.6 Operation of the DEXTU Instruction



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

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

msbd	lsb	msb	pos	size	Instruction	Comment
$32 \leq msbd < 64$	$0 \leq lsb < 32$	$32 \leq msb < 64$	$0 \leq pos < 32$	$32 < size \leq 64$	DEXTM	The field is larger than 32 bits and starts in the right-most word of the doubleword

Restrictions:

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

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

Operation:

```

lsb ← lsbminus32 + 32
if ((lsb + msbd + 1) > 64) then
    UNPREDICTABLE
endif
GPR[rt] ← 063-(msbd+1) || GPR[rs]msbd+lsb..pos

```

Exceptions:

Reserved Instruction

Programming Notes

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

31	26 25	21 20	16 15	11 10	6	5	4	3	2	0
	COP0 0100 00	MFMCO 01 011	rt	12 0110 0	0 000 00	sc 0	0 0 0	0 000		

6 5 5 5 5 1 2 3

Format: DI
DI rt

MIPS32 Release 2
MIPS32 Release 2

Purpose:

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

Description: $\text{GPR}[rt] \leftarrow \text{Status}; \text{Status}_{IE} \leftarrow 0$

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

Restrictions:

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

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

Operation:

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

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

Exceptions:

Coprocessor Unusable

Reserved Instruction (Release 1 implementations)

Programming Notes:

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

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

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL3 011111	rs	rt	msb (pos+size-1)	lsb (pos)	DINS 000111	6

6 5 5 5 5 6

Format: DINS rt, rs, pos, size

MIPS64 Release 2

Purpose: Doubleword Insert Bit Field

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

Description: $\text{GPR}[rt] \leftarrow \text{InsertField}(\text{GPR}[rt], \text{GPR}[rs], \text{msb}, \text{lsb})$

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

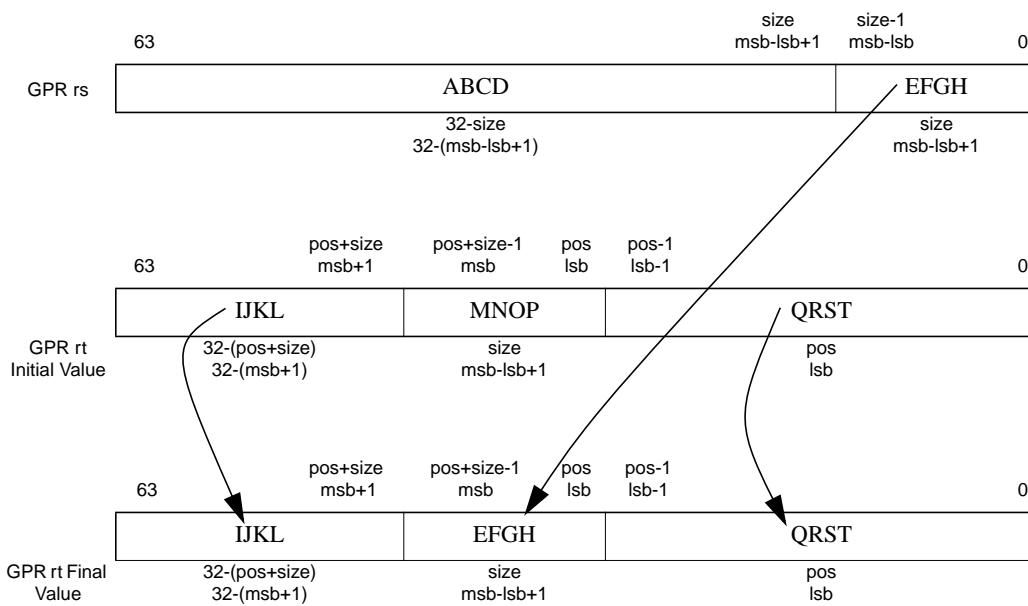
```
msb ← pos+size-1
lsb ← pos
```

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

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

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

Figure 3.7 Operation of the DINS Instruction



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

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

Restrictions:

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

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

Operation:

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

Exceptions:

Reserved Instruction

Programming Notes

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

31	26..25	21..20	16..15	11..10	6..5	0
SPECIAL3 011111	rs	rt	msbminus32 (pos+size-33)	lsb (pos)	DINSM 000101	6

Format: DINSM rt, rs, pos, size

MIPS64 Release 2

Purpose: Doubleword Insert Bit Field Middle

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

Description: $\text{GPR}[rt] \leftarrow \text{InsertField}(\text{GPR}[rt], \text{GPR}[rs], \text{msb}, \text{lsb})$

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

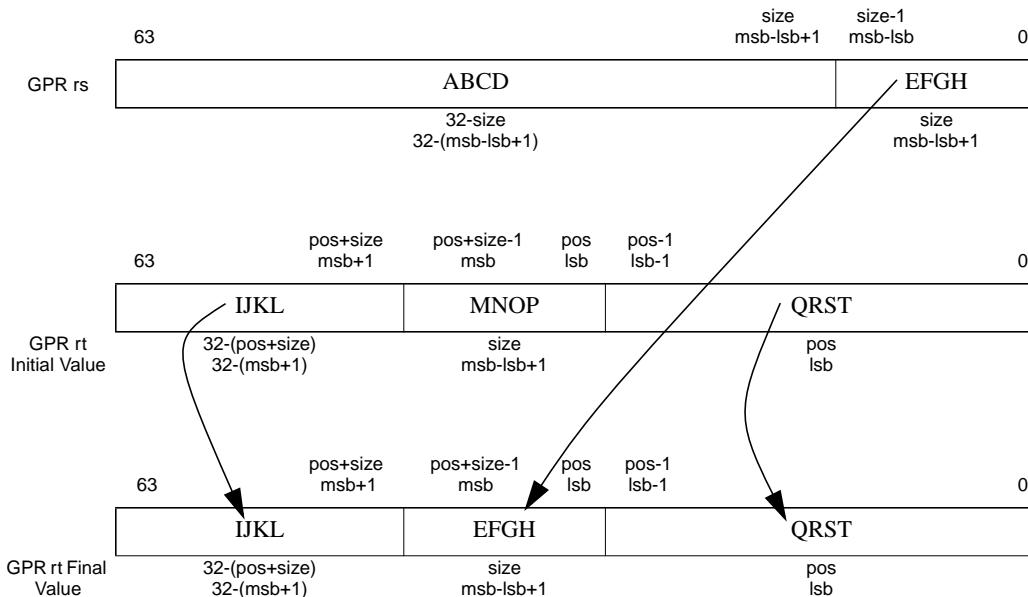
```
msbminus32 ← pos+size-1-32
lsb ← pos
msb ← msbminus32 + 32
```

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

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

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

Figure 3.8 Operation of the DINSM Instruction



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

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

Restrictions:

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

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

Operation:

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

Exceptions:

Reserved Instruction

Programming Notes

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

31	26..25	21..20	16..15	11..10	6..5	0
SPECIAL3 011111	rs	rt	msbminus32 (pos+size-33)	lsbminus32 (pos-32)	DINSU 000110	

6 5 5 5 5 6

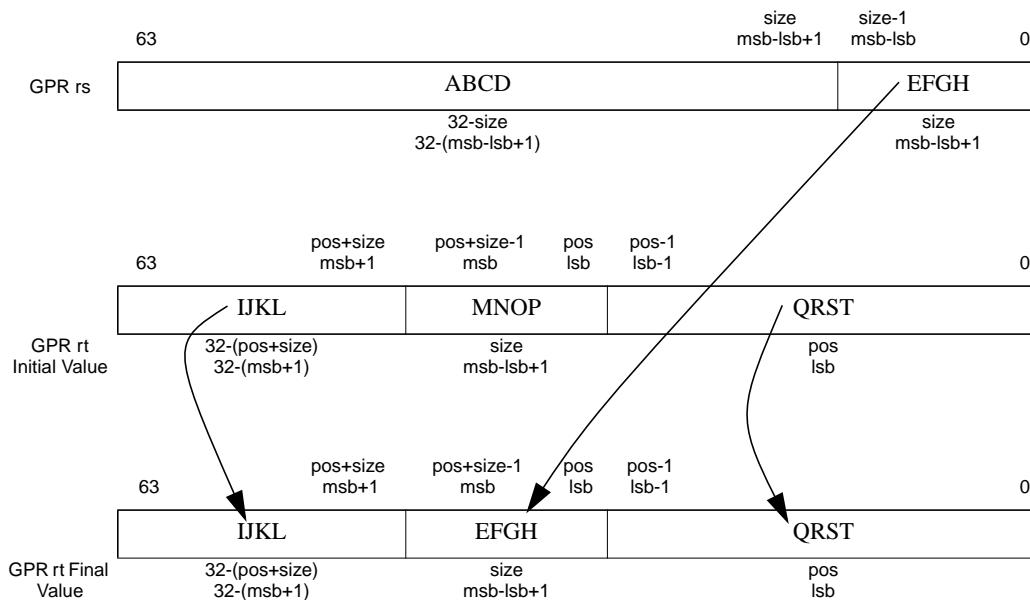
Format: DINSU rt, rs, pos, size**MIPS64 Release 2****Purpose:** Doubleword Insert Bit Field UpperTo merge a right-justified bit field from GPR *rs* into a specified position in GPR *rt*.**Description:** GPR[rt] \leftarrow InsertField(GPR[rt], GPR[rs], msb, lsb)

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

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

For this instruction, the values of *pos* and *size* must satisfy all of the following relations:
$$\begin{aligned} 32 \leq pos < 64 \\ 1 \leq size \leq 32 \\ 32 < pos+size \leq 64 \end{aligned}$$

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

Figure 3.9 Operation of the DINSU Instruction

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

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

Restrictions:

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

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

Operation:

```

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

```

Exceptions:

Reserved Instruction

Programming Notes

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

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

Format: DIV rs, rt

MIPS32

Purpose: Divide Word

To divide a 32-bit signed integers

Description: $(HI, LO) \leftarrow GPR[rs] / GPR[rt]$

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

No arithmetic exception occurs under any circumstances.

Restrictions:

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

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

Operation:

```

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
    UNPREDICTABLE
endif
q  ← GPR[rs]31..0 div GPR[rt]31..0
LO ← sign_extend(q31..0)
r  ← GPR[rs]31..0 mod GPR[rt]31..0
HI ← sign_extend(r31..0)

```

Exceptions:

None

Programming Notes:

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

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

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

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

ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the divide so that other instructions can execute in parallel.

Historical Perspective:

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

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	ft	fs	fd	DIV 000011	6

Format: DIV(fmt)
 DIV.S fd, fs, ft
 DIV.D fd, fs, ft

MIPS32
 MIPS32

Purpose: Floating Point Divide

To divide FP values

Description: $FPR[fd] \leftarrow FPR[fs] / FPR[ft]$

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

Restrictions:

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

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

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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

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

Format: DIVU rs, rt**MIPS32****Purpose:** Divide Unsigned Word

To divide a 32-bit unsigned integers

Description: $(HI, LO) \leftarrow GPR[rs] / GPR[rt]$

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

No arithmetic exception occurs under any circumstances.

Restrictions:

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

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

Operation:

```

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
    UNPREDICTABLE
endif
q ← (0 || GPR[rs]31..0) div (0 || GPR[rt]31..0)
r ← (0 || GPR[rs]31..0) mod (0 || GPR[rt]31..0)
LO ← sign_extend(q31..0)
HI ← sign_extend(r31..0)

```

Exceptions:

None

Programming Notes:

See “Programming Notes” for the [DIV](#) instruction.

Historical Perspective:

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

31	26 25	21 20	16 15	11 10	3 2	0
COP0 010000	DMF 00001	rt	rd	0 0000 0000	sel	

Format: DMFC0 rt, rd
DMFC0 rt, rd, sel

MIPS64
MIPS64

Purpose:

Doubleword Move from Coprocessor 0

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

Description: GPR[rt] \leftarrow CPR[0,rd,sel]

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

Restrictions:

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

Operation:

```
datadoubleword  $\leftarrow$  CPR[0,rd,sel]
GPR[rt]  $\leftarrow$  datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

31	26 25	21 20	16 15	11 10	0
COP1 010001	DMF 000001	rt	fs	0 000 0000 0000	11

Format: DMFC1 rt,fs**MIPS64****Purpose:** Doubleword Move from Floating Point

To move a doubleword from an FPR to a GPR.

Description: GPR[rt] \leftarrow FPR[fs]

The contents of FPR fs are loaded into GPR rt.

Restrictions:**Operation:**

```
datadoubleword  $\leftarrow$  ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)
GPR[rt]  $\leftarrow$  datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Historical Information:

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

31	26 25	21 20	16 15	0
COP2 010010	DMF 00001	rt	Impl	

Format: DMFC2 rt, rd
DMFC2, rt, rd, sel

MIPS64
MIPS64

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

Purpose: Doubleword Move from Coprocessor 2

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

Description: GPR[rt] \leftarrow CP2CPR[Impl]

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

Restrictions:

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

Operation:

```
datadoubleword  $\leftarrow$  CP2CPR[Impl]
GPR[rt]  $\leftarrow$  datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

31	26 25	21 20	16 15	11 10	3 2	0
COP0 010000	DMT 00101	rt	rd	0 0000 0000	sel	

6 5 5 5 8 3

Format: DMTC0 rt, rd
DMTC0 rt, rd, sel

MIPS64
MIPS64

Purpose: Doubleword Move to Coprocessor 0

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

Description: CPR[0,rd,sel] \leftarrow GPR[rt]

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

Restrictions:

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

Operation:

```
datadoubleword  $\leftarrow$  GPR[rt]
CPR[0,rd,sel]  $\leftarrow$  datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

31	26 25	21 20	16 15	11 10	0
COP1 010001	DMT 00101	rt	fs	0 000 0000 0000	11

Format: DMTC1 rt, fs**MIPS64****Purpose:** Doubleword Move to Floating Point

To copy a doubleword from a GPR to an FPR

Description: FPR[fs] \leftarrow GPR[rt]The doubleword contents of GPR *rt* are placed into FPR *fs*.**Restrictions:****Operation:**

```
datadoubleword  $\leftarrow$  GPR[rt]
StoreFPR(fs, UNINTERPRETED_DOUBLEWORD, datadoubleword)
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

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

31	26 25	21 20	16 15	0
COP2 010010	DMT 00101	rt	Impl	

6 5 5 16

Format: DMTC2 rt, *Impl*
 DMTC2 rt, *Impl*, sel

MIPS64
MIPS64

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

Purpose: Doubleword Move to Coprocessor 2

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

Description: CPR[2, rd, sel] \leftarrow GPR[rt]

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

Restrictions:

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

Operation:

```
datadoubleword  $\leftarrow$  GPR[rt]
CP2CPR[Impl]  $\leftarrow$  datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

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

Format: DMULT rs, rt

MIPS64

Purpose: Doubleword Multiply

To multiply 64-bit signed integers

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

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

No arithmetic exception occurs under any circumstances.

Restrictions:

Operation:

```
prod ← GPR[rs] × GPR[rt]
LO ← prod63..0
HI ← prod127..64
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Programs that require overflow detection must check for it explicitly.

Historical Perspective:

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

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

Format: DMULTU rs, rt**MIPS64****Purpose:** Doubleword Multiply Unsigned

To multiply 64-bit unsigned integers

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

The 64-bit doubleword value in GPR *rt* is multiplied by the 64-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 128-bit result. The low-order 64-bit doubleword of the result is placed into special register *LO*, and the high-order 64-bit doubleword is placed into special register *HI*. No arithmetic exception occurs under any circumstances.

Restrictions:**Operation:**

```

prod ← (0 | GPR[rs]) × (0 | GPR[rt])
LO ← prod63..0
HI ← prod127..64

```

Exceptions:

Reserved Instruction

Programming Notes:

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

Programs that require overflow detection must check for it explicitly.

Historical Perspective:

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

31	26 25	22 21 20	16 15	11 10	6 5	0
SPECIAL 000000	0000	R 1	rt	rd	sa	DSRL 111010

6 4 1 5 5 5 6

Format: DROTR rd, rt, sa**MIPS64 Release 2****Purpose:** Doubleword Rotate Right

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

Description: $\text{GPR}[rd] \leftarrow \text{GPR}[rt] \leftrightarrow (\text{right}) \text{ sa}$

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

Restrictions:**Operation:**

$$\begin{aligned} s &\leftarrow 0 \mid\mid sa \\ \text{GPR}[rd] &\leftarrow \text{GPR}[rt]_{s-1..0} \mid\mid \text{GPR}[rt]_{63..s} \end{aligned}$$

Exceptions:

Reserved Instruction

31	26 25	22 21 20	16 15	11 10	6 5	0
SPECIAL 000000	0000	R 1	rt	rd	saminus32	DROTR32 111110

6 4 1 5 5 5 6

Format: DROTR32 rd, rt, sa**MIPS64 Release 2****Purpose:** Doubleword Rotate Right Plus 32

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

Description: $GPR[rd] \leftarrow GPR[rt] \leftrightarrow (\text{right}) (saminus32+32)$ The 64-bit doubleword contents of GPR *rt* are rotated right; the result is placed in GPR *rd*. The bit-rotate amount in the range 32 to 63 is specified by *saminus32*+32.**Restrictions:****Operation:**

$$\begin{aligned} s &\leftarrow 1 \mid\mid sa \quad /* 32+saminus32 */ \\ GPR[rd] &\leftarrow GPR[rt]_{s-1..0} \mid\mid GPR[rt]_{63..s} \end{aligned}$$

Exceptions:

Reserved Instruction

31	26 25	21 20	16 15	11 10	7 6 5	0
SPECIAL 000000	rs	rt	rd	0000	R 1	DSRLV 010110

6 5 5 5 4 1 6

Format: DROTRV rd, rt, rs**MIPS64 Release 2****Purpose:** Doubleword Rotate Right Variable

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

Description: GPR[rd] \leftarrow GPR[rt] \leftrightarrow (right) GPR[rs]

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

Restrictions:**Operation:**

$$\begin{aligned} s &\leftarrow \text{GPR}[rs]_{5..0} \\ \text{GPR}[rd] &\leftarrow \text{GPR}[rt]_{s-1..0} \quad || \quad \text{GPR}[rt]_{63..s} \end{aligned}$$

Exceptions:

Reserved Instruction

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

Format: DSBH rd, rt**MIPS64 Release 2****Purpose:** Doubleword Swap Bytes Within Halfwords

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

Description: $\text{GPR}[\text{rd}] \leftarrow \text{SwapBytesWithinHalfwords}(\text{GPR}[\text{rt}])$

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

Restrictions:

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

Operation:

```
GPR[rd] ← GPR[t]55..48 || GPR[t]63..56 || GPR[t]39..32 || GPR[t]47..40 ||  
GPR[t]23..16 || GPR[t]31..24 || GPR[t]7..0 || GPR[t]15..8
```

Exceptions:

Reserved Instruction

Programming Notes:

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

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

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

Format: DSHD rd, rt**MIPS64 Release 2****Purpose:** Doubleword Swap Halfwords Within DoublewordsTo swap the halfwords of GPR *rt* and store the value into GPR *rd*.**Description:** GPR[rd] \leftarrow SwapHalfwordsWithinDoublewords(GPR[rt])The halfwords of GPR *rt* are swapped and stored in GPR *rd*.**Restrictions:**

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

Operation:GPR[rd] \leftarrow GPR[rt]_{15..0} || GPR[rt]_{31..16} || GPR[rt]_{47..32} || GPR[rt]_{63..48}**Exceptions:**

Reserved Instruction

Programming Notes:

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

```

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

```

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL 111000	

Format: DSLL rd, rt, sa**MIPS64****Purpose:** Doubleword Shift Left Logical

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

Description: GPR[rd] \leftarrow GPR[rt] \ll saThe 64-bit doubleword contents of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit-shift amount in the range 0 to 31 is specified by *sa*.**Restrictions:****Operation:**

$$\begin{aligned} s &\leftarrow 0 \mid\mid sa \\ GPR[rd] &\leftarrow GPR[rt]_{(63-s)\dots0} \mid\mid 0^s \end{aligned}$$

Exceptions:

Reserved Instruction

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL32 111100	

6 5 5 5 5 6

Format: DSLL32 rd, rt, sa**MIPS64****Purpose:** Doubleword Shift Left Logical Plus 32

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

Description: GPR[rd] \leftarrow GPR[rt] \ll (sa+32)

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

Restrictions:**Operation:**

$$\begin{aligned} s &\leftarrow 1 \mid\mid sa \quad /* 32+sa */ \\ \text{GPR}[rd] &\leftarrow \text{GPR}[rt]_{(63-s)\dots0} \mid\mid 0^s \end{aligned}$$

Exceptions:

Reserved Instruction

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	DSLLV 010100	

6 5 5 5 5 6

Format: DSLLV rd, rt, rs**MIPS64****Purpose:** Doubleword Shift Left Logical Variable

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

Description: GPR[rd] \leftarrow GPR[rt] \ll GPR[rs]

The 64-bit doubleword contents of GPR rt are shifted left, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit-shift amount in the range 0 to 63 is specified by the low-order 6 bits in GPR rs.

Restrictions:**Operation:**

$$\begin{aligned} s &\leftarrow \text{GPR}[rs]_{5..0} \\ \text{GPR}[rd] &\leftarrow \text{GPR}[rt]_{(63-s)..0} \mid\mid 0^s \end{aligned}$$

Exceptions:

Reserved Instruction

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA 111011	

6 5 5 5 5 6

Format: DSRA rd, rt, sa**MIPS64****Purpose:** Doubleword Shift Right Arithmetic

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

Description: GPR[rd] \leftarrow GPR[rt] \gg sa (arithmetic)

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

Restrictions:**Operation:**

$$\begin{aligned} s &\leftarrow 0 \mid\mid sa \\ \text{GPR}[rd] &\leftarrow (\text{GPR}[rt]_{63})^s \mid\mid \text{GPR}[rt]_{63..s} \end{aligned}$$

Exceptions:

Reserved Instruction

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA32 111111	

Format: DSRA32 rd, rt, sa**MIPS64****Purpose:** Doubleword Shift Right Arithmetic Plus 32

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

Description: $GPR[rd] \leftarrow GPR[rt] \gg (sa+32)$ (arithmetic)The doubleword contents of GPR *rt* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR *rd*. The bit-shift amount in the range 32 to 63 is specified by *sa*+32.**Restrictions:****Operation:**

$$\begin{aligned} s &\leftarrow 1 \mid\mid sa \quad /* 32+sa */ \\ GPR[rd] &\leftarrow (GPR[rt]_{63})^s \mid\mid GPR[rt]_{63..s} \end{aligned}$$

Exceptions:

Reserved Instruction

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

6 5 5 5 5 6

Format: DSRAV rd, rt, rs**MIPS64****Purpose:** Doubleword Shift Right Arithmetic Variable

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

Description: GPR[rd] \leftarrow GPR[rt] \gg GPR[rs] (arithmetic)

The doubleword contents of GPR rt are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR rd. The bit-shift amount in the range 0 to 63 is specified by the low-order 6 bits in GPR rs.

Restrictions:**Operation:**

$$\begin{aligned} s &\leftarrow \text{GPR}[rs]_{5..0} \\ \text{GPR}[rd] &\leftarrow (\text{GPR}[rt]_{63})^s \mid\mid \text{GPR}[rt]_{63..s} \end{aligned}$$

Exceptions:

Reserved Instruction

31	26 25	22 21 20	16 15	11 10	6 5	0
SPECIAL 000000	0000	R 0	rt	rd	sa	DSRL 111010

Format: DSRL rd, rt, sa**MIPS64****Purpose:** Doubleword Shift Right Logical

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

Description: GPR[rd] \leftarrow GPR[rt] \gg sa (logical)The doubleword contents of GPR *rt* are shifted right, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit-shift amount in the range 0 to 31 is specified by *sa*.**Restrictions:****Operation:**

$$\begin{array}{ll} s & \leftarrow 0 \mid\mid sa \\ \text{GPR}[rd] & \leftarrow 0^s \mid\mid \text{GPR}[rt]_{63..s} \end{array}$$

Exceptions:

Reserved Instruction

31	26 25	22 21 20	16 15	11 10	6 5	0
SPECIAL 000000	0000	R 0	rt	rd	saminus32	DSRL32 111110

6 1 1 5 5 5 6

Format: DSRL32 rd, rt, sa**MIPS64****Purpose:** Doubleword Shift Right Logical Plus 32

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

Description: GPR[rd] \leftarrow GPR[rt] \gg (saminus32+32) (logical)

The 64-bit doubleword contents of GPR rt are shifted right, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit-shift amount in the range 32 to 63 is specified by saminus32+32.

Restrictions:**Operation:**

$$\begin{aligned} s &\leftarrow 1 \mid\mid sa \quad /* 32+saminus32 */ \\ \text{GPR}[rd] &\leftarrow 0^s \mid\mid \text{GPR}[rt]_{63..s} \end{aligned}$$

Exceptions:

Reserved Instruction

31	26 25	21 20	16 15	11 10	7 6 5	0
SPECIAL 000000	rs	rt	rd	0000	R 0	DSRLV 010110

6 5 5 5 4 1 6

Format: DSRLV rd, rt, rs**MIPS64****Purpose:** Doubleword Shift Right Logical Variable

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

Description: GPR[rd] \leftarrow GPR[rt] \gg GPR[rs] (logical)

The 64-bit doubleword contents of GPR rt are shifted right, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit-shift amount in the range 0 to 63 is specified by the low-order 6 bits in GPR rs.

Restrictions:**Operation:**

$$\begin{aligned} s &\leftarrow \text{GPR}[rs]_{5..0} \\ \text{GPR}[rd] &\leftarrow 0^s \mid\mid \text{GPR}[rt]_{63..s} \end{aligned}$$

Exceptions:

Reserved Instruction

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	DSUB 101110	

6 5 5 5 5 6

Format: DSUB rd, rs, rt**MIPS64****Purpose:** Doubleword Subtract

To subtract 64-bit integers; trap on overflow

Description: GPR[rd] \leftarrow GPR[rs] - GPR[rt]

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

Restrictions:**Operation:**

```

temp  $\leftarrow$  (GPR[rs]63 || GPR[rs]) - (GPR[rt]63 || GPR[rt])
if (temp64  $\neq$  temp63) then
    SignalException(IntegerOverflow)
else
    GPR[rd]  $\leftarrow$  temp63..0
endif

```

Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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

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

6 5 5 5 5 6

Format: DSUBU rd, rs, rt**MIPS64****Purpose:** Doubleword Subtract Unsigned

To subtract 64-bit integers

Description: $GPR[rd] \leftarrow GPR[rs] - GPR[rt]$ The 64-bit doubleword value in GPR *rt* is subtracted from the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:**Operation: 64-bit processors** $GPR[rd] \leftarrow GPR[rs] - GPR[rt]$ **Exceptions:**

Reserved Instruction

Programming Notes:

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

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	0 00000	0 00000	0 00000	3 00011	SLL 000000	6

6 5 5 5 5 6

Format: EHB**MIPS32 Release 2****Purpose:** Execution Hazard Barrier

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

Description:

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

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

Restrictions:

None

Operation:

ClearExecutionHazards()

Exceptions:

None

Programming Notes:

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

31	26 25	21 20	16 15	11 10	6	5	4	3	2	0
COP0 0100 00	MFMCO 01 011	rt	12 0110 0	0 000 00	sc 1	0 0 0	0 000			

6 5 5 5 5 1 2 3

Format: EI
EI rt

MIPS32 Release 2
MIPS32 Release 2

Purpose:

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

Description: $\text{GPR}[rt] \leftarrow \text{Status}; \text{Status}_{IE} \leftarrow 1$

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

Restrictions:

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

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

Operation:

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

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

Exceptions:

Coprocessor Unusable

Reserved Instruction (Release 1 implementations)

Programming Notes:

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

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

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

6 1 19 6

Format: ERET**MIPS32****Purpose:** Exception Return

To return from interrupt, exception, or error trap.

Description:

ERET clears execution and instruction hazards, conditionally restores SRSCtl_{CSS} from SRSCtl_{ΠΣΣ} in a Release 2 implementation, and returns to the interrupted instruction at the completion of interrupt, exception, or error processing. ERET does not execute the next instruction (i.e., it has no delay slot).

Restrictions:

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

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

ERET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNC1 instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the ERET returns.

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

Operation:

```

if StatusERL = 1 then
    temp ← ErrorEPC
    StatusERL ← 0
else
    temp ← EPC
    StatusEXL ← 0
    if (ArchitectureRevision ≥ 2) and (SRSCtlHSS > 0) and (StatusBEV = 0) then
        SRSCtlCSS ← SRSCtlPSS
    endif
endif
if IsMIPS16Implemented() | (Config3ISA > 0) then
    PC ← temp63..1 || 0
    ISAMode ← temp0
else
    PC ← temp
endif
LLbit ← 0
ClearHazards()

```

Exceptions:

Coprocessor Unusable Exception

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL3 011111	rs	rt	msbd (size-1)	lsb (pos)	EXT 000000	6

Format: EXT rt, rs, pos, size

MIPS32 Release 2

Purpose: Extract Bit Field

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

Description: $\text{GPR}[rt] \leftarrow \text{ExtractField}(\text{GPR}[rs], \text{msbd}, \text{lsb})$

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

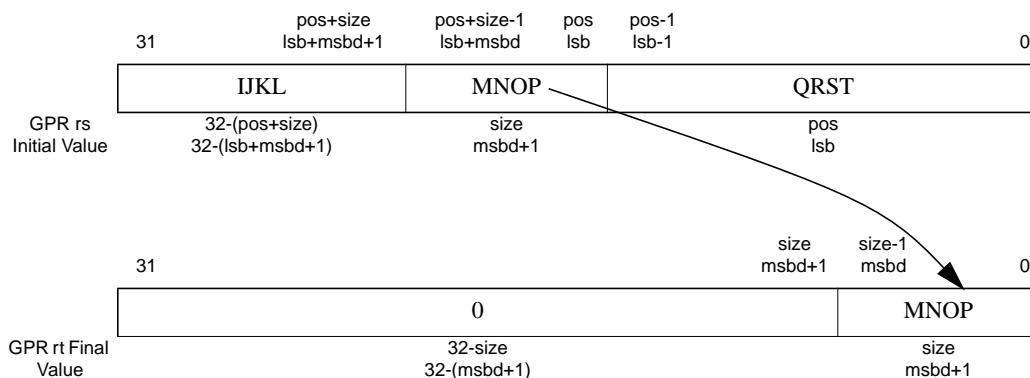
```
msbd ← size-1
lsb ← pos
```

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

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

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

Figure 3.10 Operation of the EXT Instruction



Restrictions:

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

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

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

Operation:

```
if ((lsb + msbd) > 31) or (NotWordValue(GPR[rs])) then
```

```
UNPREDICTABLE
endif
temp ← sign_extend(032-(msbd+1) || GPR[rs]msbd+lsb..lsb)
GPR[rt] ← temp
```

Exceptions:

Reserved Instruction

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

6 5 5 5 5 6

Format: FLOOR.L fmt
 FLOOR.L.S fd, fs
 FLOOR.L.D fd, fs

MIPS64, MIPS32 Release 2
 MIPS64, MIPS32 Release 2

Purpose: Floating Point Floor Convert to Long Fixed Point

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

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

The value in FPR *fs*, in format *fmt*, is converted to a value in 64-bit long fixed point format and rounded toward $-\infty$ (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^{63} to $2^{63}-1$, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation Enable bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}-1$, is written to *fd*.

Restrictions:

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

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

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

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

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

Format: FLOOR.W(fmt)FLOOR.W.S fd, fs
FLOOR.W.D fd, fsMIPS32
MIPS32**Purpose:** Floating Point Floor Convert to Word Fixed Point

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

Description: `FPR[fd] ← convert_and_round(FPR[fs])`The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format and rounded toward $-\infty$ (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

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL3 011111	rs	rt	msb (pos+size-1)	lsb (pos)	INS 000100	

6 5 5 5 5 6

Format: INS rt, rs, pos, size

MIPS32 Release 2

Purpose: Insert Bit Field

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

Description: GPR[rt] \leftarrow InsertField(GPR[rt], GPR[rs], msb, lsb)

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

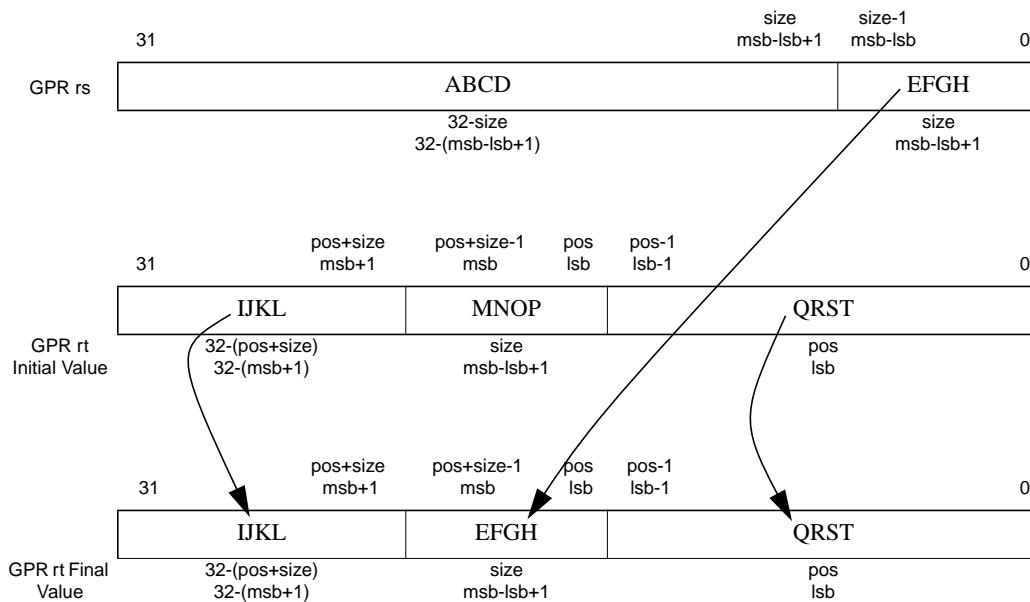
```
msb  $\leftarrow$  pos+size-1
lsb  $\leftarrow$  pos
```

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

```
0  $\leq$  pos  $<$  32
0  $<$  size  $\leq$  32
0  $<$  pos+size  $\leq$  32
```

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

Figure 3.11 Operation of the INS Instruction



Restrictions:

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

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

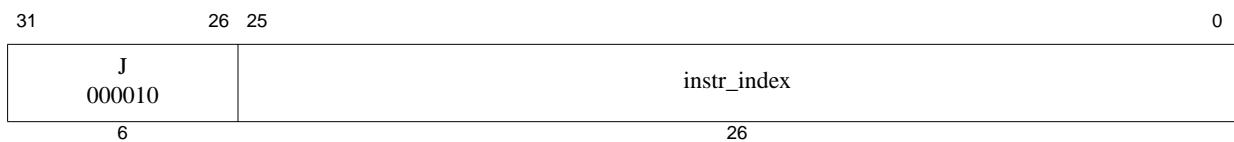
If either GPR rs or GPR rt does not contain sign-extended 32-bit values (bits **63..31** equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

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

Exceptions:

Reserved Instruction



Format: J target

MIPS32

Purpose: Jump

To branch within the current 256 MB-aligned region

Description:

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

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

Restrictions:

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: $PC \leftarrow PC_{GPRLEN-1..28} || instr_index || 0^2$

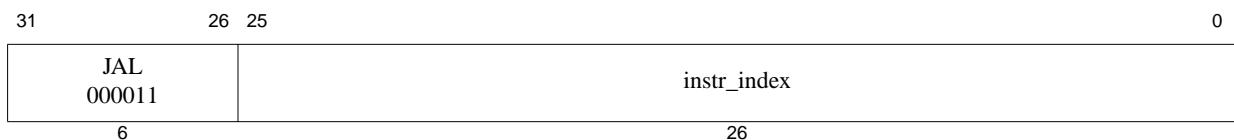
Exceptions:

None

Programming Notes:

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

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

**Format:** JAL target**MIPS32****Purpose:** Jump and Link

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

Description:

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

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

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

Restrictions:

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 ← PCGPRLEN-1..28 || instr_index || 02
```

Exceptions:

None

Programming Notes:

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

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

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

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:

GPR[rd] \leftarrow return_addr, PC \leftarrow GPR[rs]

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

For processors that do not implement the MIPS16e ASE nor microMIPS32/64 ISA:

- Jump to the effective target address in GPR rs. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

For processors that do implement the MIPS16e ASE or microMIPS32/64 ISA:

- Jump to the effective target address in GPR rs. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself. Set the *ISA Mode* bit to the value in GPR rs bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

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

Restrictions:

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

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR rs.

For processors that do not implement the microMIPS32/64 ISA, the effective target address in GPR rs must be naturally-aligned. For processors that do not implement the MIPS16e ASE nor microMIPS32/64 ISA, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction.

For processors that do implement the MIPS16e ASE or microMIPS32/64 ISA, if target ISAMode bit is 0 (GPR rs bit 0) is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

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 \leftarrow GPR[rs]
GPR[rd] \leftarrow PC + 8

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

Exceptions:

None

Programming Notes:

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

31	26 25	21 20	16 15	11 10 9	6 5	0
SPECIAL 000000	rs	0 00000	rd	1	Any other legal hint value	JALR 001001

6 5 5 5 1 4 6

Format: JALR.HB rs (rd = 31 implied)
JALR.HB rd, rs

MIPS32 Release 2
MIPS32 Release 2

Purpose: Jump and Link Register with Hazard Barrier

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

Description: GPR[rd] \leftarrow return_addr, PC \leftarrow GPR[rs], clear execution and instruction hazards

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

For processors that do not implement the MIPS16 ASE nor microMIPS32/64 ISA:

- Jump to the effective target address in GPR rs. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

For processors that do implement the MIPS16 ASE or microMIPS32/64 ISA:

- Jump to the effective target address in GPR rs. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself. Set the *ISA Mode* bit to the value in GPR rs bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

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

This instruction clears both execution and instruction hazards. Refer to the [EHB](#) instruction description for the method of clearing execution hazards alone.

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

Restrictions:

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

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR rs.

For processors that do not implement the microMIPS32/64 ISA, the effective target address in GPR rs must be naturally-aligned. For processors that do not implement the MIPS16 ASE nor microMIPS32/64 ISA, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction.

For processors that do implement the MIPS16 ASE or microMIPS32/64 ISA, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

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

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

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

Operation:

```

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

```

Exceptions:

None

Programming Notes:

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

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

Example: Clearing hazards due to an ASID change

```

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

```


31	26 25	0
JALX 011101	instr_index	26

Format: JALX target**MIPS64 with (microMIPS64 or MIPS16e)****Purpose:** Jump and Link Exchange

To execute a procedure call within the current 256 MB-aligned region and change the *ISA Mode* from MIPS64 to microMIPS64 or MIPS16e.

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call. The value stored in GPR 31 bit 0 reflects the current value of the *ISA Mode* bit.

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

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

Restrictions:

This instruction only supports 32-bit aligned branch target addresses.

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

If the microMIPS base architecture is not implemented and the MIPS16e ASE is not implemented, a Reserved Instruction Exception is initiated.

Operation:

```
I:      GPR[31] ← PC + 8
I+1:    PC ← PCGPRLEN-1..28 || instr_index || 02
        ISAMode ← (not ISAMode)
```

Exceptions:

None

Programming Notes:

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

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

31	26 25	21 20	11 10	6 5	0
SPECIAL 000000	rs	0 00 0000 0000	hint	JR 001000	6

6 5 10 5 6

Format: JR rs**MIPS32****Purpose:** Jump Register

To execute a branch to an instruction address in a register

Description: PC \leftarrow GPR[rs]

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

For processors that implement the MIPS16e ASE or microMIPS32/64 ISA, set the *ISA Mode* bit to the value in GPR rs bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one**Restrictions:**

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 of GPR rs.

For processors that do not implement the microMIPS ISA, the effective target address in GPR rs must be naturally-aligned. For processors that do not implement the MIPS16e ASE or microMIPS ISA, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction.

For processors that do implement the MIPS16e ASE or microMIPS ISA, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

In release 1 of the architecture, the only defined hint field value is 0, which sets default handling of JR. In Release 2 of the architecture, bit 10 of the hint field is used to encode an instruction hazard barrier. See the **JR.HB** instruction description for additional information.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  $\leftarrow$  GPR[rs]
I+1:if Config1CA = 0 then
    PC  $\leftarrow$  temp
else
    PC  $\leftarrow$  tempGPRLEN-1..1 || 0
    ISAMode  $\leftarrow$  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.

31	26 25	21 20	11 10 9	6 5	0
SPECIAL 000000	rs	0 00 0000 0000	1	Any other legal hint value	JR 001000

6 5 10 1 4 6

Format: JR.HB rs**MIPS32 Release 2****Purpose:** Jump Register with Hazard Barrier

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

Description: $PC \leftarrow GPR[rs]$, clear execution and instruction hazards

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

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

This instruction clears both execution and instruction hazards. Refer to the [EHB](#) instruction description for the method of clearing execution hazards alone.

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

For processors that implement the MIPS16e ASE or microMIPS32/64 ISA, set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

Restrictions:

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR *rs*.

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

For processors that do implement the MIPS16 ASE or microMIPS ISA, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

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

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

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: $temp \leftarrow GPR[rs]$

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

Exceptions:

None

Programming Notes:

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

Example: Clearing hazards due to an ASID change

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

Example: Making a write to the instruction stream visible

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

Example: Clearing instruction hazards in-line

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

31	26 25	21 20	16 15	0
LB 100000	base	rt	offset	

6 5 5 16

Format: LB rt, offset(base)**MIPS32****Purpose:** Load Byte

To load a byte from memory as a signed value

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

None

Operation:

```

vAddr  $\leftarrow$  sign_extend(offset) + GPR[base]
(pAddr, CCA)  $\leftarrow$  AddressTranslation (vAddr, DATA, LOAD)
pAddr  $\leftarrow$  pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
memdoubleword  $\leftarrow$  LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte  $\leftarrow$  vAddr2..0 xor BigEndianCPU3
GPR[rt]  $\leftarrow$  sign_extend(memdoubleword7+8*byte..8*byte)

```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

31	26 25	21 20	16 15		7 6 5	0
SPECIAL3 011111	base	rt		offset	0	LBE 101100

6 5 5 9 1 6

Format: LBE rt, offset(base)**MIPS32****Purpose:** Load Byte EVA

To load a byte as a signed value from user mode virtual address space when executing in kernel mode.

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

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

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

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

Restrictions:

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

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
memdoubleword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr2..0 xor BigEndianCPU3
GPR[rt] ← sign_extend(memdoubleword7+8*byte..8*byte)
```

Exceptions:

TLB Refill

TLB Invalid

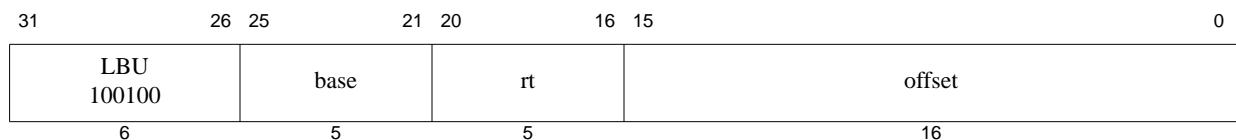
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



Format: LBU rt, offset(base)

MIPS32

Purpose: Load Byte Unsigned

To load a byte from memory as an unsigned value

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

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..3 || (pAddr2..0 xor ReverseEndian3)
memdoubleword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr2..0 xor BigEndianCPU3
GPR[rt] ← zero_extend(memdoubleword7+8*byte..8*byte)

```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

31	26 25	21 20	16 15	7 6 5	0
SPECIAL3 011111	base	rt	offset	0	LBUE 101000

6 5 5 9 1 6

Format: LBUE rt, offset(base)

MIPS32

Purpose:

To load a byte as an unsigned value from user mode virtual address space when executing in kernel mode.

Description:

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR *rt*. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

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

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

Restrictions:

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

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
memdoubleword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr2..0 xor BigEndianCPU3
GPR[rt] ← zero_extend(memdoubleword7+8*byte..8*byte)
```

Exceptions:

- TLB Refill
- TLB Invalid
- Bus Error
- Address Error
- Watch
- Reserved Instruction
- Coprocessor Unusable

31	26 25	21 20	16 15	0
LD 110111	base	rt	offset	

6 5 5 16

Format: LD rt, offset(base)**MIPS64****Purpose:** Load Doubleword

To load a doubleword from memory

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

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```

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

```

Exceptions:

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

31	26 25	21 20	16 15	0
LDC1 110101	base	ft	offset	

6 5 5 16

Format: LDC1 ft, offset(base)**MIPS32****Purpose:** Load Doubleword to Floating Point

To load a doubleword from memory to an FPR

Description: $FPR[ft] \leftarrow \text{memory}[GPR[\text{base}] + \text{offset}]$ 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 EffectiveAddress_{2..0} ≠ 0 (not doubleword-aligned).**Operation:**

```

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

```

Exceptions:

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

31	26 25	21 20	16 15	0
LDC2 110110	base	rt	offset	

6 5 5 16

Format: LDC2 rt, offset(base)**MIPS32****Purpose:** Load Doubleword to Coprocessor 2

To load a doubleword from memory to a Coprocessor 2 register

Description: CPR[2,rt,0] \leftarrow memory[GPR[base] + offset]

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

Restrictions:

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

Operation:

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

Exceptions:

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

31	26 25	21 20	16 15	0
LDL 011010	base	rt	offset	

6 5 5 16

Format: LDL rt, offset(base)**MIPS64****Purpose:** Load Doubleword Left

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

Description: GPR[rt] \leftarrow GPR[rt] MERGE memory[GPR[base] + offset]

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 8 consecutive bytes forming a doubleword (*DW*) in memory, starting at an arbitrary byte boundary.

A part of *DW*, the most-significant 1 to 8 bytes, is in the aligned doubleword containing *EffAddr*. This part of *DW* is loaded appropriately into the most-significant (left) part of GPR *rt*, leaving the remainder of GPR *rt* unchanged.

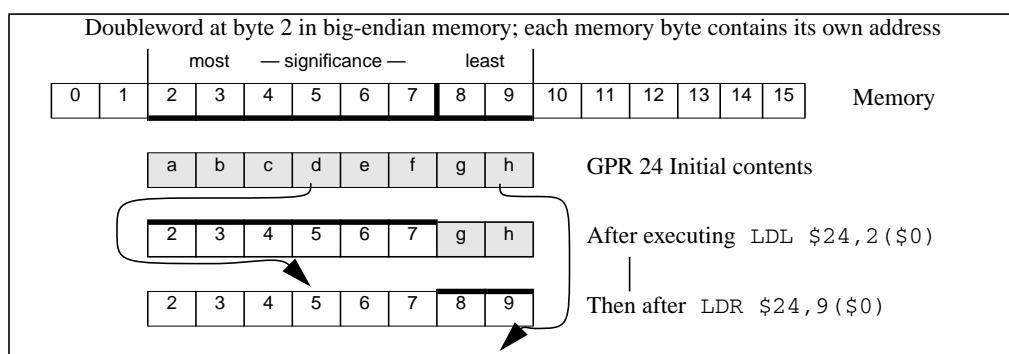
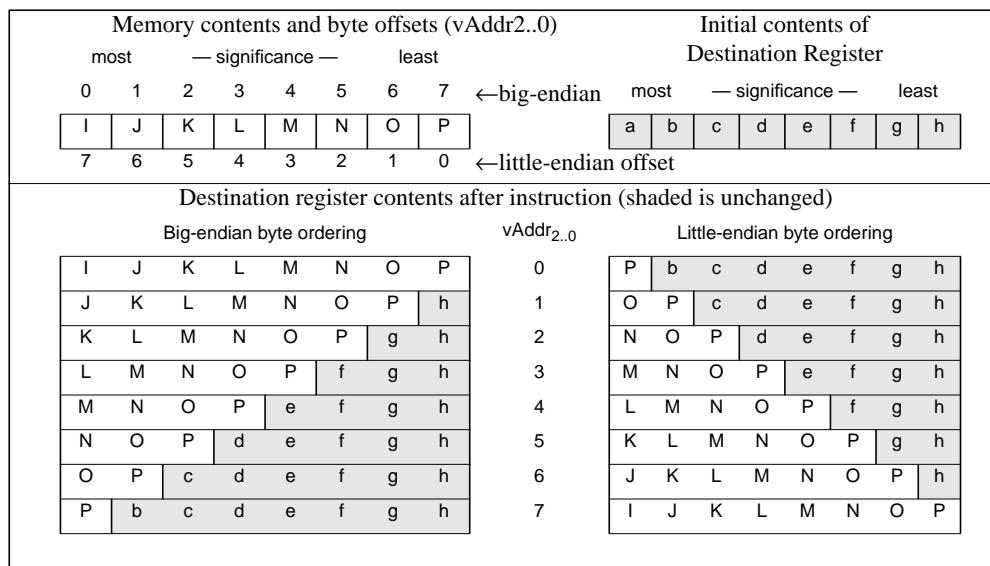
Figure 3.12 Unaligned Doubleword Load Using LDL and LDR

Figure 3-11 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 6 bytes, is located in the aligned doubleword starting with the most-significant byte at 2. LDL first loads these 6 bytes into the left part of the destination register and leaves the remainder of the destination unchanged. The complementary LDR next loads the remainder of the unaligned doubleword.

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned doubleword—the low 3 bits of the address (*vAddr2..0*)—and the current byte-ordering mode of the processor (big- or little-endian). Figure 3-12 shows the bytes loaded for every combination of offset and byte ordering.

Figure 3.13 Bytes Loaded by LDL Instruction

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

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
if BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..3 || 03
endif
byte ← vAddr2..0 xor BigEndianCPU3
memdoubleword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
GPR[rt] ← memdblword7+8*byte..0 || GPR[rt]55-8*byte..0

```

Exceptions:

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

31	26 25	21 20	16 15	0
LDR 011011	base	rt	offset	

6 5 5 16

Format: LDR rt, offset(base)**MIPS64****Purpose:** Load Doubleword Right

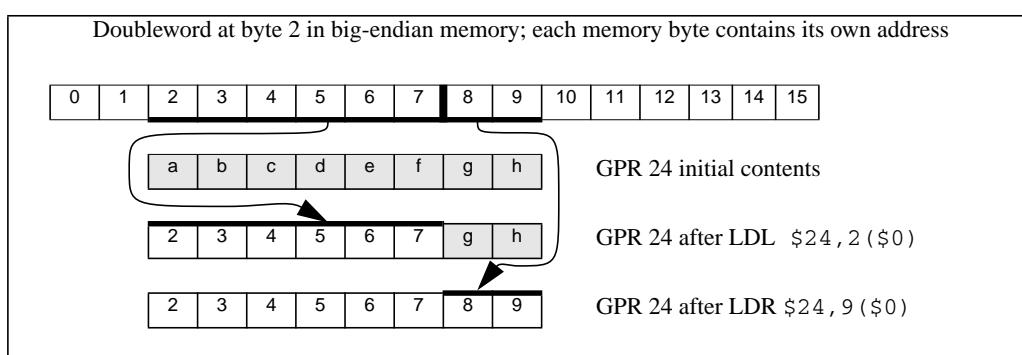
To load the least-significant part of a doubleword from an unaligned memory address

Description: GPR[rt] \leftarrow GPR[rt] MERGE memory[GPR[base] + offset]

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of 8 consecutive bytes forming a doubleword (*DW*) in memory, starting at an arbitrary byte boundary.

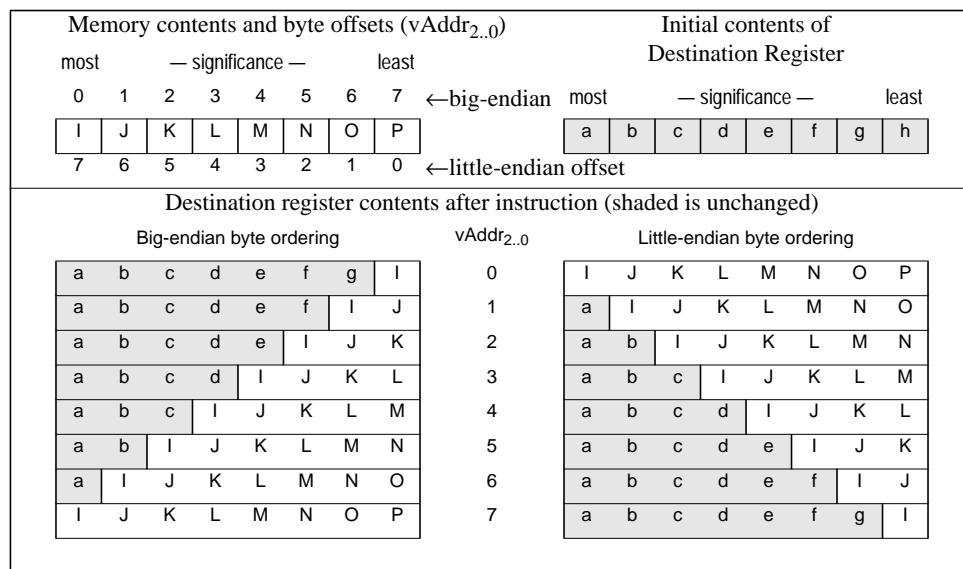
A part of *DW*, the least-significant 1 to 8 bytes, is in the aligned doubleword containing *EffAddr*. This part of *DW* is loaded appropriately into the least-significant (right) part of GPR *rt* leaving the remainder of GPR *rt* unchanged.

Figure 3-13 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. Two bytes of the *DW* are located in the aligned doubleword containing the least-significant byte at 9. LDR first loads these 2 bytes into the right part of the destination register, and leaves the remainder of the destination unchanged. The complementary LDL next loads the remainder of the unaligned doubleword.

Figure 3.14 Unaligned Doubleword Load Using LDR and LDL

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

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

Figure 3.15 Bytes Loaded by LDR Instruction**Restrictions:****Operation: 64-bit processors**

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
if BigEndianMem = 1 then
    pAddr ← pAddrPSIZE-1..3 || 03
endif
byte ← vAddr2..0 xor BigEndianCPU3
memdoubleword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
GPR[rt] ← GPR[rt]63..64-8*byte || memdoubleword63..8*byte

```

Exceptions:

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

31	26 25	21 20	16 15	11 10	6 5	0
COP1X 010011	base	index	0 00000	fd	LDXC1 000001	6

6 5 5 5 5 6

Format: LDXC1 fd, index(base)**MIPS64**
MIPS32 Release 2**Purpose:** Load Doubleword Indexed to Floating Point

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

Description: FPR[fd] \leftarrow memory[GPR[base] + GPR[index]]The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in FPR *fd*. The contents of GPR *index* and GPR *base* are added to form the effective address.**Restrictions:**An Address Error exception occurs if EffectiveAddress_{2..0} \neq 0 (not doubleword-aligned).**Operation:**

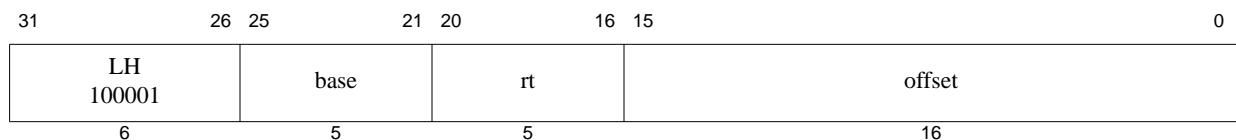
```

vAddr  $\leftarrow$  GPR[base] + GPR[index]
if vAddr2..0  $\neq$  03 then
    SignalException(AddressError)
endif
(pAddr, CCA)  $\leftarrow$  AddressTranslation (vAddr, DATA, LOAD)
memdoubleword  $\leftarrow$  LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
StoreFPR(fd, UNINTERPRETED_DOUBLEWORD, memdoubleword)

```

Exceptions:

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

**Format:** LH rt, offset(base)**MIPS32****Purpose:** Load Halfword

To load a halfword from memory as a signed value

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

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

Restrictions:

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

Operation:

```

vAddr  $\leftarrow$  sign_extend(offset) + GPR[base]
if vAddr0  $\neq$  0 then
    SignalException(AddressError)
endif
(pAddr, CCA)  $\leftarrow$  AddressTranslation(vAddr, DATA, LOAD)
pAddr  $\leftarrow$  pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian2 || 0))
memdoubleword  $\leftarrow$  LoadMemory(CCA, HALFWORD, pAddr, vAddr, DATA)
byte  $\leftarrow$  vAddr2..0 xor (BigEndianCPU2 || 0)
GPR[rt]  $\leftarrow$  sign_extend(memdoubleword15+8*byte..8*byte)

```

Exceptions:

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

31	26 25	21 20	16 15		7 6 5	0
SPECIAL3 011111	base	rt		offset	0	LHE 101101

6 5 5 9 1 6

Format: LHE rt, offset(base)**MIPS32****Purpose:** Load Halfword EVA

To load a halfword as a signed value from user mode virtual address space when executing in kernel mode.

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

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, and placed in GPR *rt*. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

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

Implementation of this instruction is specified by the *Config5_EVA* field being set to one.

Restrictions:

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

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  $\leftarrow$  sign_extend(offset) + GPR[base]
(pAddr, CCA)  $\leftarrow$  AddressTranslation (vAddr, DATA, LOAD)
pAddr  $\leftarrow$  pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian2 || 0))
memdoubleword  $\leftarrow$  LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte  $\leftarrow$  vAddr2..0 xor (BigEndianCPU2 || 0)
GPR[rt]  $\leftarrow$  sign_extend(memdoubleword15+8*byte..8*byte)
```

Exceptions:

TLB Refill

TLB Invalid

Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable

31	26 25	21 20	16 15	0
LHU 100101	base	rt	offset	

6 5 5 16

Format: LHU rt, offset(base)**MIPS32****Purpose:** Load Halfword Unsigned

To load a halfword from memory as an unsigned value

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

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..3 || (pAddr2..0 xor (ReverseEndian2 || 0))
memdoubleword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr2..0 xor (BigEndianCPU2 || 0)
GPR[rt] ← zero_extend(memdoubleword15+8*byte..8*byte)

```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

31	26 25	21 20	16 15		7 6 5	0
SPECIAL3 011111	base	rt		offset	0	LHUE 101001

6 5 5 9 1 6

Format: LHUE rt, offset(base)**MIPS32****Purpose:** Load Halfword Unsigned EVA

To load a halfword as an unsigned value from user mode virtual address space when executing in kernel mode.

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

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

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

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

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

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]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian2 || 0))
memdoubleword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr2..0 xor (BigEndianCPU2 || 0)
GPR[rt] ← zero_extend(memdoubleword15+8*byte..8*byte)

```

Exceptions:

TLB Refill

TLB Invalid

Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable

31	26 25	21 20	16 15	0
LL 110000	base	rt	offset	

6 5 5 16

Format: LL rt, offset(base)**MIPS32****Purpose:** Load Linked Word

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

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

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

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

This begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor. When an LL is executed it starts an active RMW sequence replacing any other sequence that was active. The RMW sequence is completed by a subsequent SC instruction that either completes the RMW sequence atomically and succeeds, or does not and fails.

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

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

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

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

Operation:

```

vAddr  $\leftarrow$  sign_extend(offset) + GPR[base]
if vAddr1..0  $\neq$  02 then
    SignalException(AddressError)
endif
(pAddr, CCA)  $\leftarrow$  AddressTranslation (vAddr, DATA, LOAD)
pAddr  $\leftarrow$  pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword  $\leftarrow$  LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte  $\leftarrow$  vAddr2..0 xor (BigEndianCPU || 02)
GPR[rt]  $\leftarrow$  sign_extend(memdoubleword31+8*byte..8*byte)
LLbit  $\leftarrow$  1

```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

Programming Notes:

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

31	26 25	21 20	16 15		7 6 5	0
SPECIAL3 011111	base	rt		offset	0	LLE 101110

6 5 5 9 1 6

Format: LLE rt, offset(base)**MIPS32****Purpose:** Load Linked Word EVA

To load a word from a user mode virtual address when executing in kernel mode for an atomic read-modify-write

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

The LLE and SCE instructions provide the primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations using user mode virtual addresses while executing in kernel mode.

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

This begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor. When an LLE is executed it starts an active RMW sequence replacing any other sequence that was active. The RMW sequence is completed by a subsequent SCE instruction that either completes the RMW sequence atomically and succeeds, or does not and fails.

Executing LLE on one processor does not cause an action that, by itself, causes an SCE for the same block to fail on another processor.

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

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

Implementation of this instruction is specified by the *Config5_{EVA}* field being set to one.**Restrictions:**The addressed location must be synchronizable by all processors and I/O devices sharing the location; if it is not, the result is **UNPREDICTABLE**. Which storage is synchronizable is a function of both CPU and system implementations. See the documentation of the **SCE** instruction for the formal definition.

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

Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
byte ← vAddr2..0 xor (BigEndianCPU || 02)
GPR[rt] ← sign_extend(memdoubleword31+8*byte..8*byte)

```

LLbit \leftarrow 1

Exceptions:

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

Programming Notes:

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

31	26 25	21 20	16 15	0
LLD 110100	base	rt	offset	

6 5 5 16

Format: LLD rt, offset(base)**MIPS64****Purpose:** Load Linked Doubleword

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

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

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

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

This begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor. When an LLD is executed it starts the active RMW sequence and replaces any other sequence that was active. The RMW sequence is completed by a subsequent SCD instruction that either completes the RMW sequence atomically and succeeds, or does not complete and fails.

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

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

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

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

Operation:

```

vAddr  $\leftarrow$  sign_extend(offset) + GPR[base]
if vAddr2..0  $\neq$  03 then
    SignalException(AddressError)
endif
(pAddr, CCA)  $\leftarrow$  AddressTranslation (vAddr, DATA, LOAD)
memdoubleword  $\leftarrow$  LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[rt]  $\leftarrow$  memdoubleword
LLbit  $\leftarrow$  1

```

Exceptions:

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

31	26 25	21 20	16 15	0
LUI 001111	0 00000	rt	immediate	

Format: LUI rt, immediate

MIPS32

Purpose: Load Upper Immediate

To load a constant into the upper half of a word

Description: $\text{GPR}[rt] \leftarrow \text{immediate} \mid\mid 0^{16}$

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

Restrictions:

None

Operation:

$\text{GPR}[rt] \leftarrow \text{sign_extend}(\text{immediate} \mid\mid 0^{16})$

Exceptions:

None

31	26 25	21 20	16 15	11 10	6 5	0
COP1X 010011	base	index	0 00000	fd	LUXC1 000101	

6 5 5 5 5 6 0

Format: LUXC1 fd, index(base)**MIPS64**
MIPS32 Release 2**Purpose:** Load Doubleword Indexed Unaligned to Floating Point

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

Description: $FPR[fd] \leftarrow \text{memory}[(GPR[\text{base}] + GPR[\text{index}])_{\text{PSIZE}-1..3}]$

The contents of the 64-bit doubleword at the memory location specified by the effective address are fetched and placed into the low word of FPR *fd*. The contents of GPR *index* and GPR *base* are added to form the effective address. The effective address is doubleword-aligned; EffectiveAddress_{2..0} are ignored.

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

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

Exceptions:

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

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

6 5 5 16

Format: LW rt, offset(base)**MIPS32****Purpose:** Load Word

To load a word from memory as a signed value

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

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

Restrictions:

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

Operation:

```

vAddr  $\leftarrow$  sign_extend(offset) + GPR[base]
if vAddr1..0  $\neq$  02 then
    SignalException(AddressError)
endif
(pAddr, CCA)  $\leftarrow$  AddressTranslation (vAddr, DATA, LOAD)
pAddr  $\leftarrow$  pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword  $\leftarrow$  LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte  $\leftarrow$  vAddr2..0 xor (BigEndianCPU || 02)
GPR[rt]  $\leftarrow$  sign_extend(memdoubleword31+8*byte..8*byte)

```

Exceptions:

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

31	26 25	21 20	16 15	0
LWC1 110001	base	ft	offset	

6 5 5 16

Format: LWC1 ft, offset(base)

MIPS32

Purpose: Load Word to Floating Point

To load a word from memory to an FPR

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

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 FPR *ft*. If FPRs are 64 bits wide, bits 63..32 of FPR *ft* become **UNPREDICTABLE**. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

An Address Error exception occurs if EffectiveAddress_{1..0} ≠ 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)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
StoreFPR(ft, UNINTERPRETED_WORD,
          memdoubleword31+8*bytesel..8*bytesel)

```

Exceptions:

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

31	26 25	21 20	16 15	0
LWC2 110010	base	rt	offset	

6 5 5 16

Format: LWC2 rt, offset(base)**MIPS32****Purpose:** Load Word to Coprocessor 2

To load a word from memory to a COP2 register

Description: $CPR[2, rt, 0] \leftarrow \text{memory}[GPR[\text{base}] + \text{offset}]$

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 EffectiveAddress_{1..0} ≠ 0 (not word-aligned).

Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr12..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
CPR[2, rt, 0] ← sign_extend(memdoubleword31+8*bytesel..8*bytesel)

```

Exceptions:

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

31	26 25	21 20	16 15	7 6 5	0
SPECIAL3 011111	base	rt	offset	0	LWE 101111

6 5 5 9 1 6

Format: LWE rt, offset(base)

MIPS32

Purpose: Load Word EVA

To load a word from user mode virtual address space when executing in kernel mode.

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

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

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

Implementation of this instruction is specified by the *Config5_EVA* field being set to one.

Restrictions:

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

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]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte ← vAddr2..0 xor (BigEndianCPU || 02)
GPR[rt] ← sign_extend(memdoubleword31+8*byte..8*byte)
```

Exceptions:

TLB Refill

TLB Invalid

Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable

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

6 5 5 16

Format: LWL rt, offset(base)

MIPS32

Purpose: Load Word Left

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

Description: GPR[rt] \leftarrow GPR[rt] MERGE memory[GPR[base] + offset]

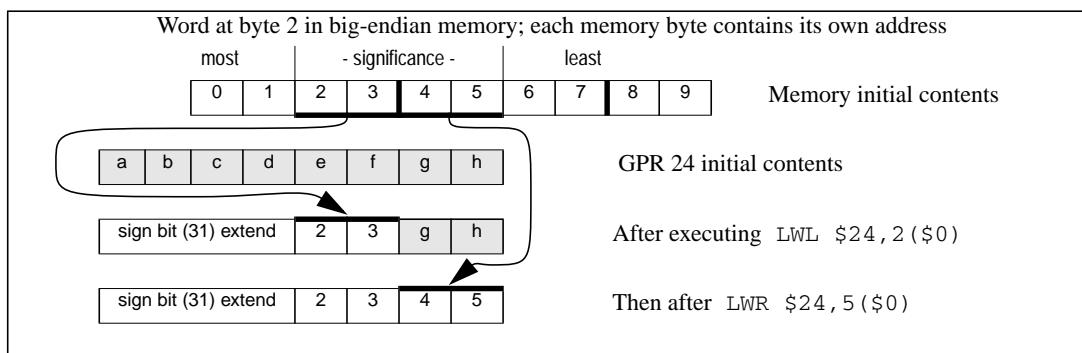
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

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

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

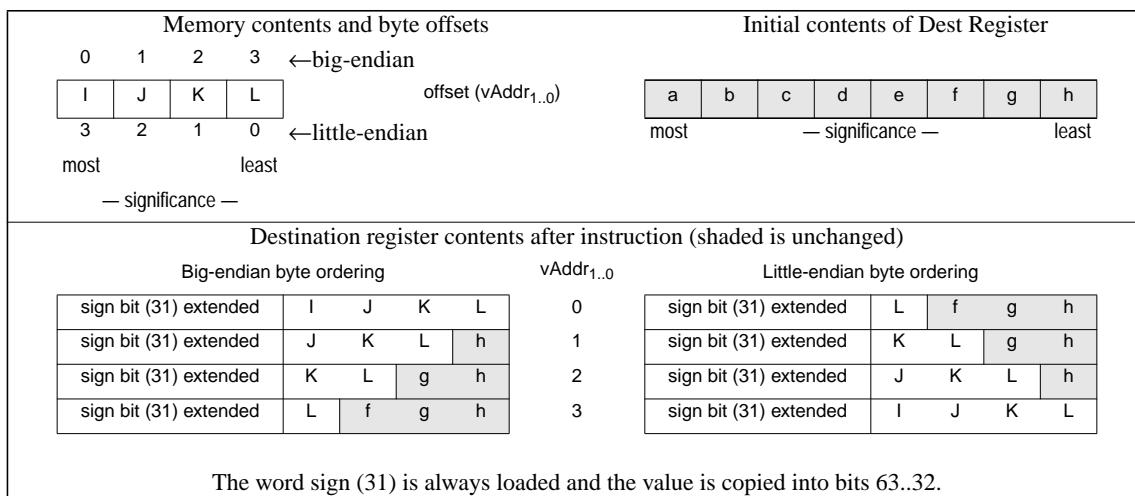
The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2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

Figure 3.16 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 3.17 Bytes Loaded by LWL Instruction



Restrictions:

None

Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
if BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..3 || 03
endif
byte ← 0 || (vAddr1..0 xor BigEndianCPU2)
word ← vAddr2 xor BigEndianCPU
memdoubleword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp ← memdoubleword31+32*word-8*byte..32*word || GPR[rt]23-8*byte..0
GPR[rt] ← (temp31)32 || temp

```

Exceptions:

None

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

Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

Historical Information:

In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.

31	26 25	21 20	16 15	7 6 5	0
SPECIAL3 011111	base	rt	offset	0	LWLE 011001

6 5 5 9 1 6

Format: LWLE rt, offset(base)

MIPS32

Purpose: Load Word Left EVA

To load the most-significant part of a word as a signed value from an unaligned user mode virtual address while executing in kernel mode.

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

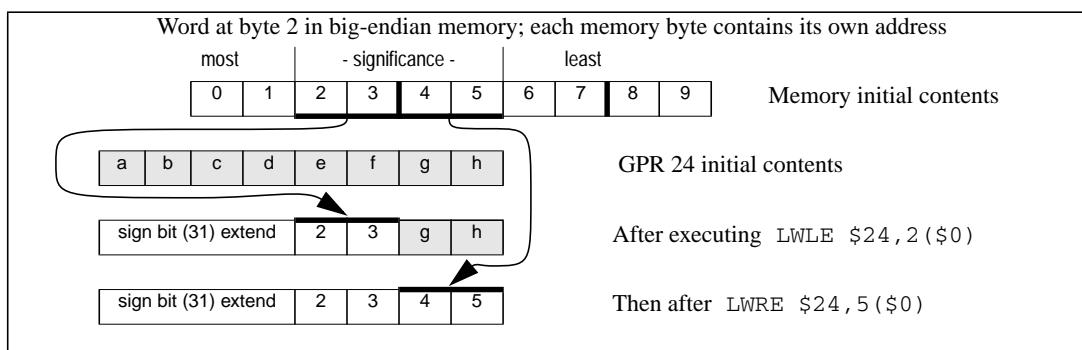
The 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

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

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

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

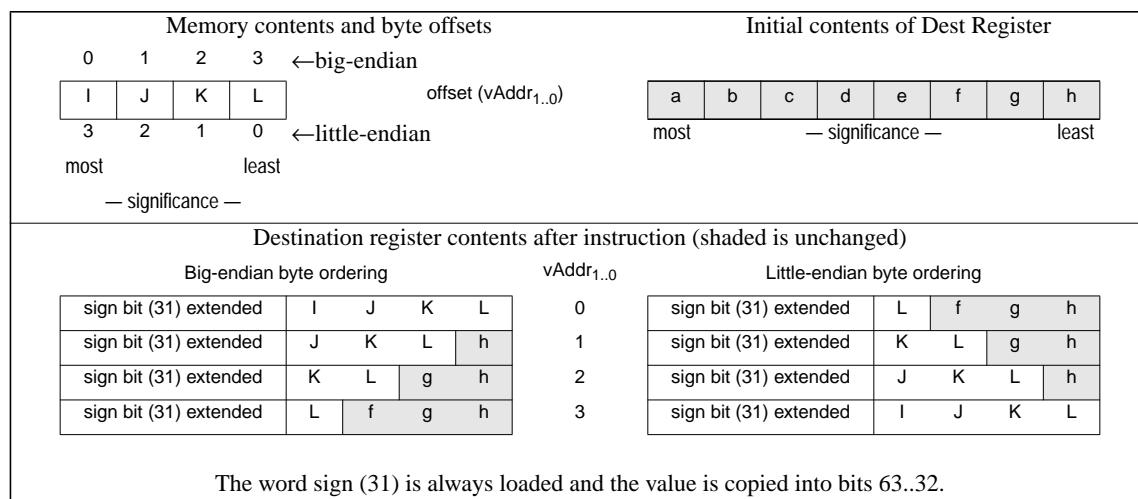
Figure 3.18 Unaligned Word Load Using LWLE and LWRE



The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address ($vAddr_{1..0}$), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.

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

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

Figure 3.19 Bytes Loaded by LWLE Instruction**Restrictions:**

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

Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
if BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..3 || 03
endif
byte ← 0 || (vAddr1..0 xor BigEndianCPU2)
word ← vAddr2 xor BigEndianCPU
memdoubleword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp ← memdoubleword31+32*word-8*byte..32*word || GPR[rt]23-8*byte..0
GPR[rt] ← (temp31)32 || temp

```

Exceptions:

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

Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

Historical Information:

In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.

31	26 25	21 20	16 15	0
LWR 100110	base	rt	offset	

6 5 5 16

Format: LWR rt, offset(base)

MIPS32

Purpose: Load Word Right

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

Description: GPR[rt] \leftarrow GPR[rt] MERGE memory[GPR[base] + offset]

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

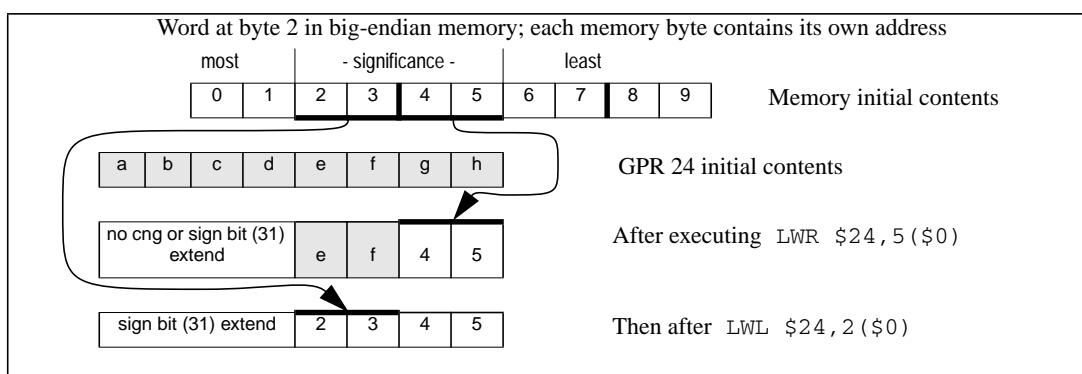
A part of *W*, the least-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. This part of *W* is loaded into the least-significant (right) part of the word in GPR *rt*. The remaining most-significant part of the word in GPR *rt* is unchanged.

If GPR *rt* is a 64-bit register, the destination word is the low-order word of the register. The loaded value is treated as a signed value; if the word sign bit (bit 31) is loaded (that is, when all 4 bytes are loaded), then the new sign bit value is copied into bits 63..32. If bit 31 is not loaded, the value of bits 63..32 is implementation dependent; the value is either unchanged or a copy of the current value of bit 31.

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

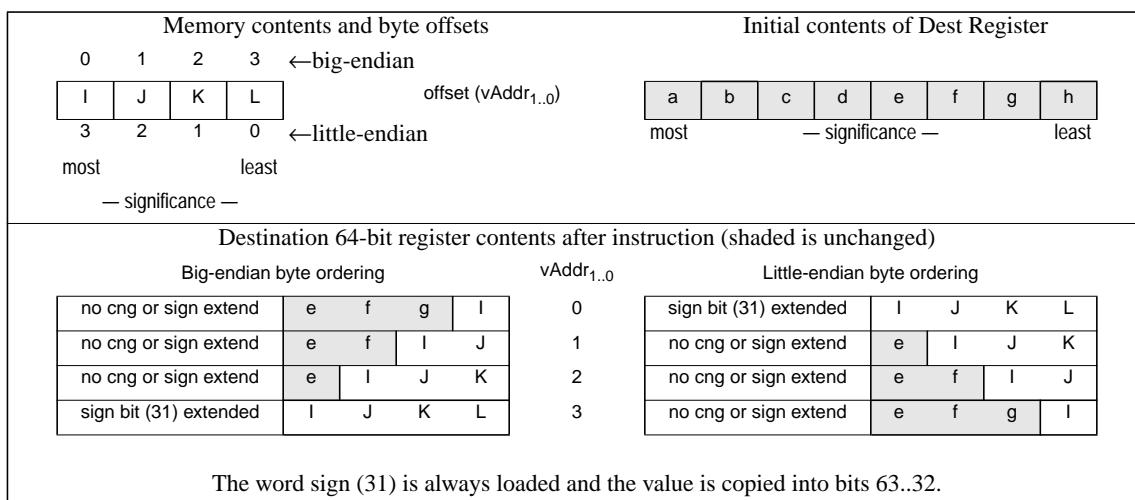
The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the least-significant byte at 5. First, LWR loads these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

Figure 3.20 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 3.21 Bytes Loaded by LWR Instruction

**Restrictions:**

None

Operation:

```

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

```

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 instruc-

tion. All such restrictions were removed from the architecture in MIPS II.

31	26 25	21 20	16 15	7 6 5	0
SPECIAL3 011111	base	rt	offset	0	LWRE 011010

Format: LWRE rt, offset(base)

MIPS32

Purpose: Load Word Right EVA

To load the least-significant part of a word from an unaligned user mode virtual memory address as a signed value while executing in kernel mode.

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

The 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

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

If GPR *rt* is a 64-bit register, the destination word is the low-order word of the register. The loaded value is treated as a signed value; if the word sign bit (bit 31) is loaded (that is, when all 4 bytes are loaded), then the new sign bit value is copied into bits 63..32. If bit 31 is not loaded, the value of bits 63..32 is implementation dependent; the value is either unchanged or a copy of the current value of bit 31.

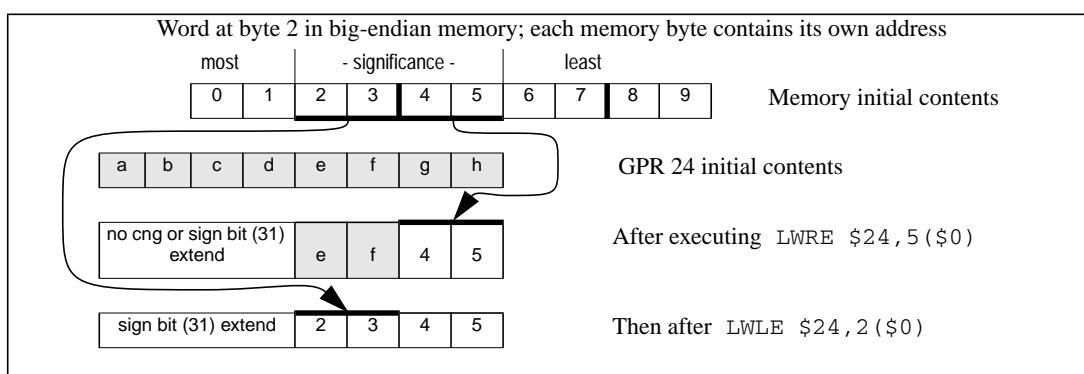
Executing both LWRE and LWLE, in either order, delivers a sign-extended word value in the destination register.

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

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

Implementation of this instruction is specified by the *Config5_EVA* field being set to one.

Figure 3.22 Unaligned Word Load Using LWLE and LWRE



The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address ($vAddr_{1..0}$), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.

Figure 3.23 Bytes Loaded by LWRE Instruction

Memory contents and byte offsets				Initial contents of Dest Register											
0	1	2	3	\leftarrow big-endian											
I	J	K	L	offset ($vAddr_{1..0}$)											
3	2	1	0	\leftarrow little-endian											
most		least													
\leftarrow significance —								a	b	c	d	e	f	g	h
								most	— significance —				least		
Destination 64-bit register contents after instruction (shaded is unchanged)															
Big-endian byte ordering				$vAddr_{1..0}$				Little-endian byte ordering							
no cng or sign extend	e	f	g	I	0	sign bit (31) extended		I	J	K	L				
no cng or sign extend	e	f	I	J	1	no cng or sign extend		e	I	J	K				
no cng or sign extend	e	I	J	K	2	no cng or sign extend		e	f	I	J				
sign bit (31) extended	I	J	K	L	3	no cng or sign extend		e	f	g	I				

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

Restrictions:

Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
if BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..3 || 03
endif
byte ← vAddr1..0 xor BigEndianCPU2
word ← vAddr2 xor BigEndianCPU
memdoubleword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp ← GPR[rt]31..32-8*byte || memdoubleword31+32*word..32*word+8*byte
if byte = 4 then
    utemp ← (temp31)32 /* loaded bit 31, must sign extend */
else
    /* one of the following two behaviors: */
    utemp ← GPR[rt]63..32 /* leave what was there alone */
    utemp ← (GPR[rt]31)32 /* sign-extend bit 31 */
endif
GPR[rt] ← utemp || temp

```

Exceptions:

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

Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

Historical Information:

In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.

31	26 25	21 20	16 15	0
LWU 100111	base	rt	offset	

6 5 5 16

Format: LWU rt, offset(base)**MIPS64****Purpose:** Load Word Unsigned

To load a word from memory as an unsigned value

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

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

Restrictions:

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

Operation:

```

vAddr  $\leftarrow$  sign_extend(offset) + GPR[base]
if vAddr1..0  $\neq$  02 then
    SignalException(AddressError)
endif
(pAddr, CCA)  $\leftarrow$  AddressTranslation (vAddr, DATA, LOAD)
pAddr  $\leftarrow$  pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword  $\leftarrow$  LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte  $\leftarrow$  vAddr2..0 xor (BigEndianCPU || 02)
GPR[rt]  $\leftarrow$  032 || memdoubleword31+8*byte..8*byte

```

Exceptions:

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

31	26 25	21 20	16 15	11 10	6 5	0
COP1X 010011	base	index	0 00000	fd	LWXC1 000000	6

6 5 5 5 5 6

Format: LWXC1 fd, index(base)**MIPS64**
MIPS32 Release 2**Purpose:** Load Word Indexed to Floating Point

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

Description: FPR[fd] \leftarrow memory[GPR[base] + GPR[index]]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of FPR_{fd}. If FPRs are 64 bits wide, bits 63..32 of FPR_{fd}s become **UNPREDICTABLE**. The contents of GPR_{index} and GPR_{base} are added to form the effective address.

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

```

vAddr  $\leftarrow$  GPR[base] + GPR[index]
if vAddr1..0  $\neq$  02 then
    SignalException(AddressError)
endif
(pAddr, CCA)  $\leftarrow$  AddressTranslation (vAddr, DATA, LOAD)
pAddr  $\leftarrow$  pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword  $\leftarrow$  LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
bytesel  $\leftarrow$  vAddr2..0 xor (BigEndianCPU || 02)
StoreFPR(fd, UNINTERPRETED_WORD,
          memdoubleword31+8*bytesel..8*bytesel)

```

Exceptions:

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

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

Format: MADD rs, rt**MIPS32****Purpose:** Multiply and Add Word to Hi,Lo

To multiply two words and add the result to Hi, Lo

Description: $(HI, LO) \leftarrow (HI, LO) + (GPR[rs] \times GPR[rt])$

The 32-bit word value in GPR *rs* is multiplied by the 32-bit word value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is added to the 64-bit concatenated values of $HI_{31..0}$ and $LO_{31..0}$. The most significant 32 bits of the result are sign-extended and written into *HI* and the least significant 32 bits are sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits **63..31** equal), then the results of the operation are **UNPREDICTABLE**.

This instruction does not provide the capability of writing directly to a target GPR.

Operation:

```

if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp <- ( $HI_{31..0} || LO_{31..0}$ ) + (GPR[rs]31..0 × GPR[rt]31..0)
HI <- sign_extend(temp63..32)
LO <- sign_extend(temp31..0)

```

Exceptions:

None

Programming Notes:

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

31	26 25	21 20	16 15	11 10	6 5	3 2	0
COP1X 010011	fr	ft	fs	fd	MADD 100	fmt	

Format: MADD.fmt

MADD.S fd, fr, fs, ft

MIPS64, MIPS32 Release 2

MADD.D fd, fr, fs, ft

MIPS64, MIPS32 Release 2

MADD.PS fd, fr, fs, ft

MIPS64, MIPS32 Release 2

Purpose: Floating Point Multiply Add

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

Description: $FPR[fd] \leftarrow (FPR[fs] \times FPR[ft]) + FPR[fr]$

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce an intermediate product. If $FIR_{Has2008}=0$ or $FCSR_{MAC2008}=0$ then the intermediate product is rounded according to the current rounding mode in *FCSR*. If $FCSR_{MAC2008}=1$ then the intermediate product is calculated to infinite precision. The value in FPR *fr* is added to the product. The result sum is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*.

If $FIR_{Has2008}=0$ or $FCSR_{MAC2008}=0$ then the results and flags are as if separate floating-point multiply and add instructions were executed. If $FCSR_{MAC2008}=1$, the multiply operation can only signal invalid operation among the IEEE exceptions.

MADD.PS multiplies then adds the upper and lower halves of FPR *fr*, FPR *fs*, and FPR *ft* independently, and ORs together any generated exceptional conditions.

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

Restrictions:

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

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

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

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL2 011100	rs	rt	0 00000	0 00000	MADDU 000001	

Format: MADDU rs, rt**MIPS32****Purpose:** Multiply and Add Unsigned Word to Hi,LoTo multiply two unsigned words and add the result to *HI*, *LO*.**Description:** $(HI, LO) \leftarrow (HI, LO) + (GPR[rs] \times GPR[rt])$

The 32-bit word value in GPR *rs* is multiplied by the 32-bit word value in GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is added to the 64-bit concatenated values of $HI_{31..0}$ and $LO_{31..0}$. The most significant 32 bits of the result are sign-extended and written into *HI* and the least significant 32 bits are sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

This instruction does not provide the capability of writing directly to a target GPR.

Operation:

```

if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp <- ( $HI_{31..0} \parallel LO_{31..0}$ ) + (( $0^{32} \parallel GPR[rs]_{31..0}$ )  $\times$  ( $0^{32} \parallel GPR[rt]_{31..0}$ ))
HI <- sign_extend(temp $_{63..32}$ )
LO <- sign_extend(temp $_{31..0}$ )

```

Exceptions:

None

Programming Notes:

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

31	26 25	21 20	16 15	11 10	3 2	0
COP0 010000	MF 00000	rt	rd	0 0000000	sel	

6 5 5 5 8 3 0

Format: MFC0 rt, rd
MFC0 rt, rd, sel

MIPS32
MIPS32

Purpose: Move from Coprocessor 0

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

Description: GPR[rt] \leftarrow CPR[0,rd,sel]

The contents of the coprocessor 0 register specified by the combination of *rd* and *sel* are sign-extended and loaded into general register *rt*. Note that not all coprocessor 0 registers support the *sel* field. In those instances, the *sel* field must be zero.

When the coprocessor 0 register specified is the *EntryLo0* or the *EntryLo1* register, the RI/XI fields are moved to bits 31:30 of the destination register. This feature supports MIPS32 backward compatibility on a MIPS64 system.

Restrictions:

The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *rd* and *sel*.

Operation:

```

reg = rd
data  $\leftarrow$  CPR[0,reg,sel]31..0
if (reg,sel = EntryLo1 or reg,sel = EntryLo0) then
    GPR[rt]29..0  $\leftarrow$  data29..0
    GPR[rt]31  $\leftarrow$  data63
    GPR[rt]30  $\leftarrow$  data62
    GPR[rt]63..32  $\leftarrow$  sign_extend(data63)
else
    GPR[rt]  $\leftarrow$  sign_extend(data)
endif

```

Exceptions:

Coprocessor Unusable

Reserved Instruction

31	26 25	21 20	16 15	11 10	0
COP1 010001	MF 00000	rt	fs	0 000 0000 0000	11

Format: MFC1 rt, fs**MIPS32****Purpose:** Move Word From Floating Point

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

Description: GPR[rt] \leftarrow FPR[fs]

The contents of FPR fs are sign-extended and loaded into general register rt.

Restrictions:**Operation:**

```

data  $\leftarrow$  ValueFPR(fs, UNINTERPRETED_WORD)31..0
GPR[rt]  $\leftarrow$  sign_extend(data)

```

Exceptions:

Coprocessor Unusable, Reserved Instruction

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

31	26 25	21 20	16 15	11 10	8 7	0
COP2 010010	MF 00000	rt			Impl	

Format: MFC2 rt, Impl
MFC2, rt, Impl, sel

MIPS32
MIPS32

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

Purpose: Move Word From Coprocessor 2

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

Description: GPR[rt] \leftarrow CP2CPR[Impl]

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

Restrictions:

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

Operation:

```
data  $\leftarrow$  CP2CPR[Impl]31..0
GPR[rt]  $\leftarrow$  sign_extend(data)
```

Exceptions:

Coprocessor Unusable

31	26 25	21 20	16 15	11 10	0
COP1 010001	MFH 00011	rt	fs	0 000 0000 0000	11

Format: MFHC1 rt, fs**MIPS32 Release 2****Purpose:** Move Word From High Half of Floating Point Register

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

Description: $GPR[rt] \leftarrow \text{sign_extend}(FPR[fs]_{63..32})$

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

Restrictions:

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

The results are **UNPREDICTABLE** if $\text{Status}_{FR} = 0$ and *fs* is odd.

Operation:

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

Exceptions:

Coprocessor Unusable

Reserved Instruction

31	26 25	21 20	16 15	11 10	3 2	0
COP2 010010	MFH 00011	rt		Impl	16	

Format: MFHC2 rt, Impl
MFHC2, rt, rd, sel

MIPS32 Release 2
MIPS32 Release 2

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

Purpose: Move Word From High Half of Coprocessor 2 Register

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

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

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

Restrictions:

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

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

Operation:

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

Exceptions:

Coprocessor Unusable

Reserved Instruction

31	26 25		16 15		11 10	6 5	0
SPECIAL 000000	0 00 0000 0000		rd		0 00000	MFHI 010000	6

6 10 5 5 6

Format: MFHI rd**MIPS32****Purpose:** Move From HI RegisterTo copy the special purpose *HI* register to a GPR**Description:** GPR[rd] ← HIThe contents of special register *HI* are loaded into GPR *rd*.**Restrictions:**

None

Operation:

GPR[rd] ← HI

Exceptions:

None

Historical Information:

In the MIPS I, II, and III architectures, the two instructions which follow the MFHI must not 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	16 15	11 10	6 5	0
SPECIAL 000000	0 00 0000 0000	rd	0 00000	MFLO 010010	

6 10 5 5 6

Format: MFLO rd**MIPS32****Purpose:** Move From LO RegisterTo copy the special purpose *LO* register to a GPR**Description:** GPR[rd] ← LOThe contents of special register *LO* are loaded into GPR *rd*.**Restrictions:**

None

Operation:

GPR[rd] ← LO

Exceptions:

None

Historical Information:

In the MIPS I, II, and III architectures, the two instructions which follow the MFLO must not modify the *Hl* register. If this restriction is violated, the result of the MFLO is **UNPREDICTABLE**. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	0 00000	fs	fd	MOV 000110	

6 5 5 5 5 6

Format: MOV.fmt
 MOV.S fd, fs
 MOV.D fd, fs
 MOV.PS fd, fs

MIPS32
 MIPS32

MIPS64, MIPS32 Release 2

Purpose: Floating Point Move

To move an FP value between FPRs

Description: FPR[fd] \leftarrow FPR[fs]

The value in FPR *fs* is placed into FPR *fd*. The source and destination are values in format *fmt*. In paired-single format, both the halves of the pair are copied to *fd*.

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

Restrictions:

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

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

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

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

31	26 25	21 20	18 17	16 15	11 10	6 5	0
SPECIAL 000000	rs	cc	0 0	tf 0	rd	0 00000	MOVF 000001

Format: MOVF rd, rs, cc**MIPS32****Purpose:** Move Conditional on Floating Point False

To test an FP condition code then conditionally move a GPR

Description: if FPConditionCode(cc) = 0 then GPR[rd] ← GPR[rs]

If the floating point condition code specified by CC is zero, then the contents of GPR rs are placed into GPR rd.

Restrictions:**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 010001	fmt	cc	0 0	tf 0	fs	fd	MOVF 010001

Format: MOVF.fmt

MOVF.S fd, fs, cc

MIPS32

MOVF.D fd, fs, cc

MIPS32

MOVF.PS fd, fs, cc

MIPS64

MIPS32 Release 2

Purpose: Floating Point Move Conditional on Floating Point False

To test an FP condition code then conditionally move an FP value

Description: if FPConditionCode(cc) = 0 then FPR[fd] ← FPR[fs]If the floating point condition code specified by CC is zero, then the value in FPR *fs* is placed into FPR *fd*. The source and destination are values in format *fmt*.If the condition code is not zero, then FPR *fs* is not copied and FPR *fd* retains its previous value in format *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes **UNPREDICTABLE**.MOVF.PS conditionally merges the lower half of FPR *fs* into the lower half of FPR *fd* if condition code *CC* is zero, and independently merges the upper half of FPR *fs* into the upper half of FPR *fd* if condition code *CC+1* is zero. The *CC* field must be even; if it is odd, the result of this operation is **UNPREDICTABLE**.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**. The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.The result of MOVF.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.**Operation:**

```

if fmt ≠ PS
    if FPConditionCode(cc) = 0 then
        StoreFPR(fd, fmt, ValueFPR(fs, fmt))
    else
        StoreFPR(fd, fmt, ValueFPR(fd, fmt))
    endif
else
    mask ← 0
    if FPConditionCode(cc+0) = 0 then mask ← mask or 0xF0 endif
    if FPConditionCode(cc+1) = 0 then mask ← mask or 0x0F endif
    StoreFPR(fd, PS, ByteMerge(mask, fd, fs))
endif

```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

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:** Move Conditional on Not Zero

To conditionally move a GPR after testing a GPR value

Description: if GPR[rt] ≠ 0 then GPR[rd] ← GPR[rs]If the value in GPR *rt* is not equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.**Restrictions:**

None

Operation:

```

if GPR[rt] ≠ 0 then
    GPR[rd] ← GPR[rs]
endif

```

Exceptions:

None

Programming Notes:

The non-zero value tested might be the *condition true* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions or a boolean value read from memory.

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	rt	fs	fd	MOVN 010011	6

Format: MOVN.fmt

MOVN.S fd, fs, rt

MIPS32

MOVN.D fd, fs, rt

MIPS32

MOVN.PS fd, fs, rt

MIPS64, MIPS32 Release 2

Purpose: Floating Point Move Conditional on Not Zero

To test a GPR then conditionally move an FP value

Description: if GPR[rt] ≠ 0 then FPR[fd] ← FPR[fs]If the value in GPR *rt* is not equal to zero, then the value in FPR *fs* is placed in FPR *fd*. The source and destination are values in format *fmt*.If GPR *rt* contains zero, then FPR *fs* is not copied and FPR *fd* contains its previous value in format *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes **UNPREDICTABLE**.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.The result of MOVN.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.**Operation:**

```

if GPR[rt] ≠ 0 then
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif

```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

31	26 25	21 20	18 17 16 15		11 10	6 5	0
SPECIAL 000000	rs	cc	0 0	tf 1	rd	0 00000	MOVCI 000001

Format: MOVT rd, rs, cc

MIPS32

Purpose: Move Conditional on Floating Point True

To test an FP condition code then conditionally move a GPR

Description: if FPConditionCode(cc) = 1 then GPR[rd] ← GPR[rs]

If the floating point condition code specified by CC is one, then the contents of GPR rs are placed into GPR rd.

Restrictions:

Operation:

```
if FPConditionCode(cc) = 1 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 010001	fmt	cc	0 0	tf 1	fs	fd	MOVF 010001

Format: MOVT.fmt

MOVT.S fd, fs, cc

MIPS32

MOVT.D fd, fs, cc

MIPS32

MOVT.PS fd, fs, cc

MIPS64, MIPS32 Release 2

Purpose: Floating Point Move Conditional on Floating Point True

To test an FP condition code then conditionally move an FP value

Description: if FPConditionCode(cc) = 1 then FPR[fd] ← FPR[fs]If the floating point condition code specified by CC is one, then the value in FPR *fs* is placed into FPR *fd*. The source and destination are values in format *fmt*.If the condition code is not one, then FPR *fs* is not copied and FPR *fd* contains its previous value in format *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes **UNPREDICTABLE**.MOVT.PS conditionally merges the lower half of FPR *fs* into the lower half of FPR *fd* if condition code CC is one, and independently merges the upper half of FPR *fs* into the upper half of FPR *fd* if condition code CC+1 is one. The CC field should be even; if it is odd, the result of this operation is **UNPREDICTABLE**.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**. The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.The result of MOVT.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.**Operation:**

```

if fmt ≠ PS
    if FPConditionCode(cc) = 0 then
        StoreFPR(fd, fmt, ValueFPR(fs, fmt))
    else
        StoreFPR(fd, fmt, ValueFPR(fd, fmt))
    endif
else
    mask ← 0
    if FPConditionCode(cc+0) = 0 then mask ← mask or 0xF0 endif
    if FPConditionCode(cc+1) = 0 then mask ← mask or 0x0F endif
    StoreFPR(fd, PS, ByteMerge(mask, fd, fs))
endif

```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

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 0

Format: MOVZ rd, rs, rt**MIPS32****Purpose:** Move Conditional on Zero

To conditionally move a GPR after testing a GPR value

Description: if GPR[rt] = 0 then GPR[rd] ← GPR[rs]If the value in GPR *rt* is equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.**Restrictions:**

None

Operation:

```

if GPR[rt] = 0 then
    GPR[rd] ← GPR[rs]
endif

```

Exceptions:

None

Programming Notes:

The zero value tested might be the *condition false* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions or a boolean value read from memory.

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	rt	fs	fd	MOVZ 010010	

6 5 5 5 5 6

Format: MOVZ . fmt

MOVZ.S fd, fs, rt

MIPS32

MOVZ.D fd, fs, rt

MIPS32

MOVZ.PS fd, fs, rt

MIPS64, MIPS32 Release 2

Purpose: Floating Point Move Conditional on Zero

To test a GPR then conditionally move an FP value

Description: if GPR[rt] = 0 then FPR[fd] ← FPR[fs]If the value in GPR *rt* is equal to zero then the value in FPR *fs* is placed in FPR *fd*. The source and destination are values in format *fmt*.If GPR *rt* is not zero, then FPR *fs* is not copied and FPR *fd* contains its previous value in format *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes **UNPREDICTABLE**.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.The result of MOVZ.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.**Operation:**

```

if GPR[rt] = 0 then
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif

```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL2 011100	rs	rt	0 00000	0 00000	MSUB 000100	

Format: MSUB rs, rt**MIPS32****Purpose:** Multiply and Subtract Word to Hi,LoTo multiply two words and subtract the result from *HI, LO***Description:** $(HI, LO) \leftarrow (HI, LO) - (GPR[rs] \times GPR[rt])$

The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values of $HI_{31..0}$ and $LO_{31..0}$. The most significant 32 bits of the result are sign-extended and written into *HI* and the least significant 32 bits are sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits **63..31** equal), then the results of the operation are **UNPREDICTABLE**.

This instruction does not provide the capability of writing directly to a target GPR.

Operation:

```

if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp <- ( $HI_{31..0} || LO_{31..0}$ ) - (GPR[rs]31..0 × GPR[rt]31..0)
HI <- sign_extend(temp63..32)
LO <- sign_extend(temp31..0)

```

Exceptions:

None

Programming Notes:

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

31	26 25	21 20	16 15	11 10	6 5	3 2	0
COP1X 010011	fr	ft	fs	fd	MSUB 101	fmt	

Format: MSUB.fmt

MSUB.S fd, fr, fs, ft

MIPS64, MIPS32 Release 2

MSUB.D fd, fr, fs, ft

MIPS64, MIPS32 Release 2

MSUB.PS fd, fr, fs, ft

MIPS64, MIPS32 Release 2

Purpose: Floating Point Multiply Subtract

To perform a combined multiply-then-subtract of FP values

Description: $FPR[fd] \leftarrow (FPR[fs] \times FPR[ft]) - FPR[fr]$

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce an intermediate product. If $FIR_{Has2008}=0$ or $FCSR_{MAC2008}=0$ then the intermediate product is rounded according to the current rounding mode in *FCSR*. If $FCSR_{MAC2008}=1$ then the intermediate product is calculated to infinite precision. The value in FPR *fr* is subtracted from the product. The subtraction result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*.

If $FIR_{Has2008}=0$ or $FCSR_{MAC2008}=0$ then the results and flags are as if separate floating-point multiply and subtract instructions were executed. If $FCSR_{MAC2008}=1$, the multiply operation can only signal invalid operation among the IEEE exceptions.

MSUB.PS multiplies then subtracts the upper and lower halves of FPR *fr*, FPR *fs*, and FPR *ft* independently, and ORs together any generated exceptional conditions.

Cause bits are ORed into the *Flag* bits if no exception is taken.

Restrictions:

The fields *fr*, *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of MSUB.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, (vfs ×fmt vft) -fmt vfr))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL2 011100	rs	rt	0 00000	0 00000	MSUBU 000101	6

Format: MSUBU rs, rt**MIPS32****Purpose:** Multiply and Subtract Word to Hi,LoTo multiply two words and subtract the result from *HI, LO***Description:** $(HI, LO) \leftarrow (HI, LO) - (GPR[rs] \times GPR[rt])$

The 32-bit word value in GPR *rs* is multiplied by the 32-bit word value in GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values of $HI_{31..0}$ and $LO_{31..0}$. The most significant 32 bits of the result are sign-extended and written into *HI* and the least significant 32 bits are sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits **63..31** equal), then the results of the operation are **UNPREDICTABLE**.

This instruction does not provide the capability of writing directly to a target GPR.

Operation:

```

if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp <- (HI31..0 || LO31..0) - ((032 || GPR[rs]31..0) × (032 || GPR[rt]31..0))
HI <- sign_extend(temp63..32)
LO <- sign_extend(temp31..0)

```

Exceptions:

None

Programming Notes:

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

31	26 25	21 20	16 15	11 10	3 2	0
COP0 010000	MT 00100	rt	rd	0 0000 000	sel	

Format: MTC0 rt, rd
MTC0 rt, rd, sel

MIPS32
MIPS32

Purpose: Move to Coprocessor 0

To move the contents of a general register to a coprocessor 0 register.

Description: CPR[0, rd, sel] \leftarrow GPR[rt]

The contents of general register rt are loaded into the coprocessor 0 register specified by the combination of rd and sel. Not all coprocessor 0 registers support the the sel field. In those instances, the sel field must be set to zero.

When the coprocessor 0 destination register specified is the *EntryLo0* or the *EntryLo1* register, bits 31:30 appear at the RI/XI fields of the destination register. This feature supports MIPS32 backward compatibility on a MIPS64 system.

Restrictions:

The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by rd and sel.

Operation:

```

data  $\leftarrow$  GPR[rt]
reg  $\leftarrow$  rd
if (reg,sel = EntryLo1 or reg,sel = EntryLo0) then
    CPR[0,reg,sel]29..0  $\leftarrow$  data29..0
    CPR[0,reg,sel]63  $\leftarrow$  data31
    CPR[0,reg,sel]62  $\leftarrow$  data30
    CPR[0,reg,sel]61:30  $\leftarrow$  032
else if (Width(CPR[0,reg,sel]) = 64) then
    CPR[0,reg,sel]  $\leftarrow$  data
else
    CPR[0,reg,sel]  $\leftarrow$  data31..0
endif

```

Exceptions:

Coprocessor Unusable

Reserved Instruction

31	26 25	21 20	16 15	11 10	0
COP1 010001	MT 00100	rt	fs	0 000 0000 0000	11

Format: MTC1 rt, fs**MIPS32****Purpose:** Move Word to Floating Point

To copy a word from a GPR to an FPU (CP1) general register

Description: FPR[fs] \leftarrow GPR[rt]The low word in GPR *rt* is placed into the low word of FPR *fs*. If FPRs are 64 bits wide, bits 63..32 of FPR *fs* become **UNPREDICTABLE**.**Restrictions:****Operation:**

```
data  $\leftarrow$  GPR[rt]31..0
StoreFPR(fs, UNINTERPRETED_WORD, data)
```

Exceptions:

Coprocessor Unusable

Historical Information:For MIPS I, MIPS II, and MIPS III the value of FPR *fs* is **UNPREDICTABLE** for the instruction immediately following MTC1.

31	26 25	21 20	16 15	11 10	8 7	0
COP2 010010	MT 00100	rt		Impl	16	

Format: MTC2 rt, Impl
MTC2 rt, Impl, sel

MIPS32
MIPS32

The syntax shown above is an example using MTC1 as a model. The specific syntax is implementation dependent.

Purpose: Move Word to Coprocessor 2

To copy a word from a GPR to a COP2 general register

Description: CP2CPR[Impl] \leftarrow GPR[rt]

The low word in GPR *rt* is placed into the low word of coprocessor 2 general register denoted by the *Impl* field. If coprocessor 2 general registers are 64 bits wide, bits 63..32 of the register denoted by the *Impl* field become **UNPREDICTABLE**. The interpretation of the *Impl* field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:

The results are **UNPREDICTABLE** if *Impl* specifies a coprocessor 2 register that does not exist.

Operation:

```
data  $\leftarrow$  GPR[rt]31..0
CP2CPR[Impl]  $\leftarrow$  data
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

31	26 25	21 20	16 15	11 10	0
COP1 010001	MTH 00111	rt	fs	0 000 0000 0000	11

Format: MTHC1 rt, fs**MIPS32 Release 2****Purpose:** Move Word to High Half of Floating Point Register

To copy a word from a GPR to the high half of an FPU (CP1) general register

Description: $FPR[fs]_{63..32} \leftarrow GPR[rt]_{31..0}$ The low word in GPR *rt* is placed into the high word of FPR *fs*. This instruction is primarily intended to support 64-bit floating point units on a 32-bit CPU, but the semantics of the instruction are defined for all cases.**Restrictions:**

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

The results are **UNPREDICTABLE** if $Status_{FR} = 0$ and *fs* is odd.**Operation:**

```

newdata ← GPR[rt]_{31..0}
olddata ← ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)_{31..0}
StoreFPR(fs, UNINTERPRETED_DOUBLEWORD, newdata || olddata)

```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Programming NotesWhen paired with MTC1 to write a value to a 64-bit FPR, the MTC1 must be executed first, followed by the MTHC1. This is because of the semantic definition of MTC1, which is not aware that software will be using an MTHC1 instruction to complete the operation, and sets the upper half of the 64-bit FPR to an **UNPREDICTABLE** value.

31	26 25	21 20	16 15	11 10	0
COP2 010010	MTH 00111	rt		Impl	

Format: MTHC2 rt, Impl
MTHC2 rt, Impl, sel

MIPS32 Release 2
MIPS32 Release 2

The syntax shown above is an example using MTHC1 as a model. The specific syntax is implementation dependent.

Purpose: Move Word to High Half of Coprocessor 2 Register

To copy a word from a GPR to the high half of a COP2 general register

Description: $CP2CPR[Impl]_{63..32} \leftarrow GPR[rt]_{31..0}$

The low word in GPR *rt* is placed into the high word of coprocessor 2 general register denoted by the *Impl* field. The interpretation of the *Impl* field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:

The results are **UNPREDICTABLE** if *Impl* specifies a coprocessor 2 register that does not exist, or if that register is not 64 bits wide.

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

```
data ← GPR[rt]31..0
CP2CPR[Impl] ← data || CPR[2,rd,sel]31..0
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Programming Notes

When paired with MTC2 to write a value to a 64-bit CPR, the MTC2 must be executed first, followed by the MTHC2. This is because of the semantic definition of MTC2, which is not aware that software will be using an MTHC2 instruction to complete the operation, and sets the upper half of the 64-bit CPR to an **UNPREDICTABLE** value.

31	26 25	21 20	15	6 5	0
SPECIAL 000000	rs		0 000 0000 0000 0000	MTHI 010001	6

Format: MTHI rs**MIPS32****Purpose:** Move to HI RegisterTo copy a GPR to the special purpose *HI* register**Description:** HI ← GPR[rs]The contents of GPR *rs* are loaded into special register *HI*.**Restrictions:**

A computed result written to the *HI/LO* pair by DIV, DIVU, DDIV, DDIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either *HI* or *LO*.

If an MTHI instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of *LO* are **UNPREDICTABLE**. The following example shows this illegal situation:

```
MULT    r2,r4  # start operation that will eventually write to HI,LO
...
        # code not containing mfhi or mflo
MTHI    r6
...
        # code not containing mflo
MFLO    r3      # this mflo would get an UNPREDICTABLE value
```

Operation:

HI ← GPR[rs]

Exceptions:

None

Historical Information:

In MIPS I-III, if either of the two preceding instructions is MFHI, the result of that MFHI is **UNPREDICTABLE**. Reads of the *HI* or *LO* special register must be separated from any subsequent instructions that write to them by two or more instructions. In MIPS IV and later, including MIPS32 and MIPS64, this restriction does not exist.

31	26 25	21 20		6 5	0
SPECIAL 000000	rs		0 000 0000 0000 0000	MTLO 010011	6

Format: MTLO rs**MIPS32****Purpose:** Move to LO RegisterTo copy a GPR to the special purpose *LO* register**Description:** LO \leftarrow GPR[rs]The contents of GPR *rs* are loaded into special register *LO*.**Restrictions:**

A computed result written to the *HI/LO* pair by DIV, DIVU, DDIV, DDIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either *HI* or *LO*.

If an MTLO instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of *HI* are **UNPREDICTABLE**. The following example shows this illegal situation:

```
MULT    r2,r4  # start operation that will eventually write to HI,LO
...
        # code not containing mfhi or mflo
MTLO    r6
...
        # code not containing mfhi
MFHI    r3      # this mfhi would get an UNPREDICTABLE value
```

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.

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL2 011100	rs	rt	rd	0 00000	MUL 000010	6

Format: MUL rd, rs, rt

MIPS32

Purpose: Multiply Word to GPR

To multiply two words and write the result to a GPR.

Description: $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}]$

The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The least significant 32 bits of the product are sign-extended and written to GPR *rd*. The contents of *HI* and *LO* are **UNPREDICTABLE** after the operation. No arithmetic exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Note that this instruction does not provide the capability of writing the result to the *HI* and *LO* registers.

Operation:

```

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
    UNPREDICTABLE
endif
temp ← GPR[rs] × GPR[rt]
GPR[rd] ← sign_extend(temp31..0)
HI ← UNPREDICTABLE
LO ← UNPREDICTABLE

```

Exceptions:

None

Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read GPR *rd* before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	ft	fs	fd	MUL 000010	

Format: MUL fmt

MUL.S fd, fs, ft

MIPS32

MUL.D fd, fs, ft

MIPS32

MUL.PS fd, fs, ft

MIPS64

MIPS32 Release 2

Purpose: Floating Point Multiply

To multiply FP values

Description: FPR[fd] \leftarrow FPR[fs] \times FPR[ft]

The value in FPR *fs* is multiplied by the value in FPR *ft*. The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. MUL.PS multiplies the upper and lower halves of FPR *fs* and FPR *ft* independently, and ORs together any generated exceptional conditions.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of MUL.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

$$\text{StoreFPR}(\text{fd}, \text{fmt}, \text{ValueFPR}(\text{fs}, \text{fmt}) \times_{\text{fmt}} \text{ValueFPR}(\text{ft}, \text{fmt}))$$
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

31	26 25	21 20	16 15	6 5	0
SPECIAL 000000	rs	rt	0 00 0000 0000	MULT 011000	6

Format: MULT rs, rt**MIPS32****Purpose:** Multiply Word

To multiply 32-bit signed integers

Description: $(HI, LO) \leftarrow GPR[rs] \times GPR[rt]$

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is sign-extended and placed into special register *LO*, and the high-order 32-bit word is sign-extended and placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
    UNPREDICTABLE
endif
prod ← GPR[rs]31..0 × GPR[rt]31..0
LO ← sign_extend(prod31..0)
HI ← sign_extend(prod63..32)

```

Exceptions:

None

Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

31	26 25	21 20	16 15	6 5	0
SPECIAL 000000	rs	rt	0 00 0000 0000	MULTU 011001	6

Format: MULTU rs, rt

MIPS32

Purpose: Multiply Unsigned Word

To multiply 32-bit unsigned integers

Description: $(HI, LO) \leftarrow GPR[rs] \times GPR[rt]$

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit word of the result is sign-extended and placed into special register *LO*, and the high-order 32-bit word is sign-extended and placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```

if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
prod ← (0 || GPR[rs]31..0) × (0 || GPR[rt]31..0)
LO ← sign_extend(prod31..0)
HI ← sign_extend(prod63..32)

```

Exceptions:

None

Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read *LO* or *HI* before the results are written interlocks until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	0 00000	fs	fd	NEG 000111	6

Format: NEG fmt
 NEG.S fd, fs
 NEG.D fd, fs
 NEG.PS fd, fs

MIPS32
 MIPS32
MIPS64, MIPS32 Release 2

Purpose: Floating Point Negate

To negate an FP value

Description: $FPR[fd] \leftarrow -FPR[fs]$

The value in FPR fs is negated and placed into FPR fd . The value is negated by changing the sign bit value. The operand and result are values in format fmt . NEG.PS negates the upper and lower halves of FPR fs independently, and ORs together any generated exceptional conditions.

If $FIR_{Has2008}=0$ or $FCSR_{ABS2008}=0$ then this operation is arithmetic. For this case, any NaN operand signals invalid operation.

If $FCSR_{ABS2008}=1$ then this operation is non-arithmetic. For this case, both regular floating point numbers and NAN values are treated alike, only the sign bit is affected by this instruction. No IEEE exception can be generated for this case.

Restrictions:

The fields fs and fd must specify FPRs valid for operands of type fmt ; if they are not valid, the result is **UNPREDICTABLE**. The operand must be a value in format fmt ; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of NEG.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
StoreFPR(fd, fmt, Negate(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

31	26 25	21 20	16 15	11 10	6 5	3 2	0
COP1X 010011	fr	ft	fs	fd	NMADD 110	fmt	

6 5 5 5 5 3 3

Format: NMADD.fmt

NMADD.S fd, fr, fs, ft

MIPS64, MIPS32 Release 2

NMADD.D fd, fr, fs, ft

MIPS64, MIPS32 Release 2

NMADD.PS fd, fr, fs, ft

MIPS64, MIPS32 Release 2

Purpose: Floating Point Negative Multiply Add

To negate a combined multiply-then-add of FP values

Description: $FPR[fd] \leftarrow -((FPR[fs] \times FPR[ft]) + FPR[fr])$

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce an intermediate product. If *FIR_{Has2008}*=0 or *FCSR_{MAC2008}*=0 then the intermediate product is rounded according to the current rounding mode in *FCSR*. If *FCSR_{MAC2008}*=1 then the intermediate product is calculated to infinite precision. The value in FPR *fr* is added to the product.

The result sum is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, negated by changing the sign bit, and placed into FPR *fd*. The operands and result are values in format *fmt*.

If *FIR_{Has2008}*=0 or *FCSR_{MAC2008}*=0 then the results and flags are as if separate floating-point multiply and add and negate instructions were executed. If *FCSR_{MAC2008}*=1, the multiply operation can only signal invalid operation among the IEEE exceptions. If *FCSR_{MAC2008}*=1, the negate portion of the instruction is arithmetic (regardless of the setting of *FCSR_{ABS2008}*) and will not invert the sign bit of QNAN result coming from the sum portion of the instruction.

NMADD.PS applies the operation to the upper and lower halves of FPR *fr*, FPR *fs*, and FPR *ft* independently, and ORs together any generated exceptional conditions.

Cause bits are ORed into the *Flag* bits if no exception is taken.

Restrictions:

The fields *fr*, *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of NMADD.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, -(vfr +fmt (vfs ×fmt vft)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

31	26 25	21 20	16 15	11 10	6 5	3 2	0
COP1X 010011	fr	ft	fs	fd	NMSUB 111	fmt	

6 5 5 5 5 3 3

Format: NMSUB.fmt

NMSUB.S fd, fr, fs, ft

MIPS64, MIPS32 Release 2

NMSUB.D fd, fr, fs, ft

MIPS64, MIPS32 Release 2

NMSUB.PS fd, fr, fs, ft

MIPS64, MIPS32 Release 2

Purpose: Floating Point Negative Multiply Subtract

To negate a combined multiply-then-subtract of FP values

Description: $FPR[fd] \leftarrow -((FPR[fs] \times FPR[ft]) - FPR[fr])$

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce an intermediate product. If *FIR_{Has2008}*=0 or *FCSR_{MAC2008}*=0 then the intermediate product is rounded according to the current rounding mode in *FCSR*. If *FCSR_{MAC2008}*=1 then the intermediate product is calculated to infinite precision. The value in FPR *fr* is subtracted from the product.

The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, negated by changing the sign bit, and placed into FPR *fd*. The operands and result are values in format *fmt*.

If *FIR_{Has2008}*=0 or *FCSR_{MAC2008}*=0 then the results and flags are as if separate floating-point multiply and subtract and negate instructions were executed. If *FCSR_{MAC2008}*=1, the multiply operation can only signal invalid operation among the IEEE exceptions. If *FCSR_{MAC2008}*=1, the negate portion of the instruction is arithmetic (regardless of the setting of *FCSR_{ABS2008}*) and will not invert the sign bit of QNAN result coming from the sum portion of the instruction.

NMSUB.PS applies the operation to the upper and lower halves of FPR *fr*, FPR *fs*, and FPR *ft* independently, and ORs together any generated exceptional conditions.

Cause bits are ORed into the *Flag* bits if no exception is taken.

Restrictions:

The fields *fr*, *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of NMSUB.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, -(vfs ×fmt vft) -fmt vfr)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

No Operation

NOP

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	0 00000	0 00000	0 00000	0 00000	SLL 000000	6

Format: NOP

Assembly Idiom

Purpose: No Operation

To perform no operation.

Description:

NOP is the assembly idiom used to denote no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 0.

Restrictions:

None

Operation:

None

Exceptions:

None

Programming Notes:

The zero instruction word, which represents SLL, r0, r0, 0, is the preferred NOP for software to use to fill branch and jump delay slots and to pad out alignment sequences.

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	NOR 100111	6

Format: NOR rd, rs, rt

MIPS32

Purpose: Not Or

To do a bitwise logical NOT OR

Description: $GPR[rd] \leftarrow GPR[rs] \text{ NOR } GPR[rt]$

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical NOR operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

$GPR[rd] \leftarrow GPR[rs] \text{ nor } GPR[rt]$

Exceptions:

None

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	OR 100101	6

Format: OR rd, rs, rt

MIPS32

Purpose: Or

To do a bitwise logical OR

Description: GPR[rd] \leftarrow GPR[rs] or GPR[rt]

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical OR operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

GPR[rd] \leftarrow GPR[rs] or GPR[rt]

Exceptions:

None

31	26 25	21 20	16 15	0
6 ORI 001101	5 rs	5 rt	immediate	16

Format: ORI rt, rs, immediate**MIPS32****Purpose:** Or Immediate

To do a bitwise logical OR with a constant

Description: GPR[rt] \leftarrow GPR[rs] or immediateThe 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] or zero_extend(immediate)**Exceptions:**

None

31	26 25 24	21 20	16 15	11 10	6 5	0
SPECIAL 000000	0 00000	0 00000	0 00000	5 00101	SLL 000000	6

Format: PAUSE**MIPS32 Release 2/MT Module****Purpose:** Wait for the LLBit to clear**Description:**

Locks implemented using the LL/SC (or LLD/SCD) instructions are a common method of synchronization between threads of control. A typical lock implementation does a load-linked instruction and checks the value returned to determine whether the software lock is set. If it is, the code branches back to retry the load-linked instruction, thereby implementing an active busy-wait sequence. The PAUSE instruction is intended to be placed into the busy-wait sequence to block the instruction stream until such time as the load-linked instruction has a chance to succeed in obtaining the software lock.

The precise behavior of the PAUSE instruction is implementation-dependent, but it usually involves descheduling the instruction stream until the LLBit is zero. In a single-threaded processor, this may be implemented as a short-term WAIT operation which resumes at the next instruction when the LLBit is zero or on some other external event such as an interrupt. On a multi-threaded processor, this may be implemented as a short term YIELD operation which resumes at the next instruction when the LLBit is zero. In either case, it is assumed that the instruction stream which gives up the software lock does so via a write to the lock variable, which causes the processor to clear the LLBit as seen by this thread of execution.

The encoding of the instruction is such that it is backward compatible with all previous implementations of the architecture. The PAUSE instruction can therefore be placed into existing lock sequences and treated as a NOP by the processor, even if the processor does not implement the PAUSE instruction.

Restrictions:

The operation of the processor is **UNPREDICTABLE** if a PAUSE instruction is placed in the delay slot of a branch or a jump.

Operation:

```
if LLBit ≠ 0 then
    EPC ← PC + 4
    DescheduleInstructionStream()
endif
```

Exceptions:

None

Programming Notes:

The PAUSE instruction is intended to be inserted into the instruction stream after an LL instruction has set the LLBit and found the software lock set. The program may wait forever if a PAUSE instruction is executed and there is no possibility that the LLBit will ever be cleared.

An example use of the PAUSE instruction is included in the following example:

```
acquire_lock:
```

```
    ll      t0, 0(a0)          /* Read software lock, set hardware lock */
    bnez   t0, acquire_lock_retry: /* Branch if software lock is taken */
    addiu  t0, t0, 1           /* Set the software lock */
    sc     t0, 0(a0)          /* Try to store the software lock */
    bnez   t0, 10f            /* Branch if lock acquired successfully */
    sync
acquire_lock_retry:
    pause                /* Wait for LLBIT to clear before retry */
    b      acquire_lock       /* and retry the operation */
    nop
10:
    Critical region code

release_lock:
    sync
    sw     zero, 0(a0)        /* Release software lock, clearing LLBIT */
                               /* for any PAUSED waiters */
```

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt 10110	ft	fs	fd	PLL 101100	6

Format: PLL.PS fd, fs, ft

MIPS64, MIPS32 Release 2

Purpose: Pair Lower Lower

To merge a pair of paired single values with realignment

Description: $FPR[fd] \leftarrow \text{lower}(FPR[fs]) \mid\mid \text{lower}(FPR[ft])$

A new paired-single value is formed by catenating the lower single of FPR *fs* (bits 31..0) and the lower single of FPR *ft* (bits 31..0).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *PS*. If they are not valid, the result is **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

`StoreFPR(fd, PS, ValueFPR(fs, PS)31..0 || ValueFPR(ft, PS)31..0)`

Exceptions:

Coprocessor Unusable, Reserved Instruction

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt 10110	ft	fs	fd	PLU 101101	6

Format: PLU.PS fd, fs, ft

MIPS64, MIPS32 Release 2

Purpose:

To merge a pair of paired single values with realignment

Description:

$FPR[fd] \leftarrow \text{lower}(FPR[fs]) \mid\mid \text{upper}(FPR[ft])$

A new paired-single value is formed by catenating the lower single of FPR *fs* (bits 31..0) and the upper single of FPR *ft* (bits 63..32).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *PS*. If they are not valid, the result is **UNPREDICTABLE**.

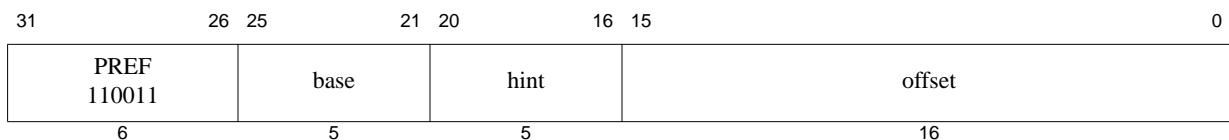
The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

`StoreFPR(fd, PS, ValueFPR(fs, PS)31..0 || ValueFPR(ft, PS)63..32)`

Exceptions:

Coprocessor Unusable, Reserved Instruction



Format: PREF hint,offset(base)

MIPS32

Purpose: Prefetch

To move data between memory and cache.

Description: prefetch_memory(GPR[base] + offset)

PREF adds the 16-bit signed *offset* to the contents of GPR *base* to form an effective byte address. The *hint* field supplies information about the way that the data is expected to be used.

PREF enables the processor to take some action, typically causing data to be moved to or from the cache, to improve program performance. The action taken for a specific PREF instruction is both system and context dependent. Any action, including doing nothing, is permitted as long as it does not change architecturally visible state or alter the meaning of a program. Implementations are expected either to do nothing, or to take an action that increases the performance of the program. The PrepareForStore function is unique in that it may modify the architecturally visible state.

PREF does not cause addressing-related exceptions, including TLB exceptions. If the address specified would cause an addressing exception, the exception condition is ignored and no data movement occurs. However even if no data is moved, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREF instruction.

PREF neither generates a memory operation nor modifies the state of a cache line for a location with an *uncached* memory access type, whether this type is specified by the address segment (e.g., kseg1), the programmed cacheability and coherency attribute of a segment (e.g., the use of the K0, KU, or K23 fields in the *Config* register), or the per-page cacheability and coherency attribute provided by the TLB.

If PREF results in a memory operation, the memory access type and cacheability&coherency attribute used for the operation are determined by the memory access type and cacheability&coherency attribute of the effective address, just as it would be if the memory operation had been caused by a load or store to the effective address.

For a cached location, the expected and useful action for the processor is to prefetch a block of data that includes the effective address. The size of the block and the level of the memory hierarchy it is fetched into are implementation specific.

In coherent multiprocessor implementations, if the effective address uses a coherent Cacheability and Coherency Attribute (CCA), then the instruction causes a coherent memory transaction to occur. This means a prefetch issued on one processor can cause data to be evicted from the cache in another processor.

The PREF instruction and the memory transactions which are sourced by the PREF instruction, such as cache refill or cache writeback, obey the ordering and completion rules of the SYNC instruction.

Table 3.33 Values of *hint* Field for PREF Instruction

Value	Name	Data Use and Desired Prefetch Action
0	load	Use: Prefetched data is expected to be read (not modified). Action: Fetch data as if for a load.

Table 3.33 Values of *hint* Field for PREF Instruction

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.”
8-20	Reserved	Reserved for future use - not available to implementations.
21-24	Implementation Dependent	Unassigned by the Architecture - available for implementation-dependent use.
25	writeback_invalidate (also known as “nudge”)	Use: Data is no longer expected to be used. Action: For a writeback cache, schedule a writeback of any dirty data. At the completion of the writeback, mark the state of any cache lines written back as invalid. If the cache line is not dirty, it is implementation dependent whether the state of the cache line is marked invalid or left unchanged. If the cache line is locked, no action is taken.
26-29	Implementation Dependent	Unassigned by the Architecture - available for implementation-dependent use.

Table 3.33 Values of *hint* Field for PREF Instruction

30	PrepareForStore	Use: Prepare the cache for writing an entire line, without the overhead involved in filling the line from memory. Action: If the reference hits in the cache, no action is taken. If the reference misses in the cache, a line is selected for replacement, any valid and dirty victim is written back to memory, the entire line is filled with zero data, and the state of the line is marked as valid and dirty. Programming Note: Because the cache line is filled with zero data on a cache miss, software must not assume that this action, in and of itself, can be used as a fast bzero-type function.
31	Implementation Dependent	Unassigned by the Architecture - available for implementation-dependent use.

Restrictions:

None

Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Bus Error, Cache Error

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:

Prefetch cannot move data to or from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. A prefetch may be used using an address pointer before the validity of the pointer is determined without worrying about an addressing exception.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREF instruction. Typically, this only occurs in systems which have high-reliability requirements.

Prefetch operations have no effect on cache lines that were previously locked with the CACHE instruction.

Hint field encodings whose function is described as “streamed” or “retained” convey usage intent from software to hardware. Software should not assume that hardware will always prefetch data in an optimal way. If data is to be truly retained, software should use the Cache instruction to lock data into the cache.

31	26 25	21 20	16 15		7 6 5	0
SPECIAL3 011111	base	hint	offset	0	PREFE 100011	6

6 5 5 9 1 6

Format: PREFE hint,offset(base)

MIPS32

Purpose: Prefetch EVA

To move data between user mode virtual address space memory and cache while operating in kernel mode.

Description: prefetch_memory(GPR[base] + offset)

PREFE adds the 9-bit signed *offset* to the contents of GPR *base* to form an effective byte address. The *hint* field supplies information about the way that the data is expected to be used.

PREFE enables the processor to take some action, typically causing data to be moved to or from the cache, to improve program performance. The action taken for a specific PREFE instruction is both system and context dependent. Any action, including doing nothing, is permitted as long as it does not change architecturally visible state or alter the meaning of a program. Implementations are expected either to do nothing, or to take an action that increases the performance of the program. The PrepareForStore function is unique in that it may modify the architecturally visible state.

PREFE does not cause addressing-related exceptions, including TLB exceptions. If the address specified would cause an addressing exception, the exception condition is ignored and no data movement occurs. However even if no data is moved, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREFE instruction.

PREFE neither generates a memory operation nor modifies the state of a cache line for a location with an *uncached* memory access type, whether this type is specified by the address segment (e.g., kseg1), the programmed cacheability and coherency attribute of a segment (e.g., the use of the K0, KU, or K23 fields in the *Config* register), or the per-page cacheability and coherency attribute provided by the TLB.

If PREFE results in a memory operation, the memory access type and cacheability&coherency attribute used for the operation are determined by the memory access type and cacheability&coherency attribute of the effective address, just as it would be if the memory operation had been caused by a load or store to the effective address.

For a cached location, the expected and useful action for the processor is to prefetch a block of data that includes the effective address. The size of the block and the level of the memory hierarchy it is fetched into are implementation specific.

In coherent multiprocessor implementations, if the effective address uses a coherent Cacheability and Coherency Attribute (CCA), then the instruction causes a coherent memory transaction to occur. This means a prefetch issued on one processor can cause data to be evicted from the cache in another processor.

The PREFE instruction and the memory transactions which are sourced by the PREFE instruction, such as cache refill or cache writeback, obey the ordering and completion rules of the SYNC instruction.

The PREFE instruction functions in exactly the same fashion as the PREF instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the *Config5_EVA* field being set to one.

Table 3.34 Values of *hint* Field for PREFE Instruction

Value	Name	Data Use and Desired Prefetch Action
0	load	Use: Prefetched data is expected to be read (not modified). Action: Fetch data as if for a load.
1	store	Use: Prefetched data is expected to be stored or modified. Action: Fetch data as if for a store.
2-3	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.”
8-20	Reserved	Reserved for future use - not available to implementations.
21-24	Implementation Dependent	Unassigned by the Architecture - available for implementation-dependent use.
25	writeback_invalidate (also known as “nudge”)	Use: Data is no longer expected to be used. Action: For a writeback cache, schedule a writeback of any dirty data. At the completion of the writeback, mark the state of any cache lines written back as invalid. If the cache line is not dirty, it is implementation dependent whether the state of the cache line is marked invalid or left unchanged. If the cache line is locked, no action is taken.
26-29	Implementation Dependent	Unassigned by the Architecture - available for implementation-dependent use.

Table 3.34 Values of *hint* Field for PREFE Instruction

30	PrepareForStore	Use: Prepare the cache for writing an entire line, without the overhead involved in filling the line from memory. Action: If the reference hits in the cache, no action is taken. If the reference misses in the cache, a line is selected for replacement, any valid and dirty victim is written back to memory, the entire line is filled with zero data, and the state of the line is marked as valid and dirty. Programming Note: Because the cache line is filled with zero data on a cache miss, software must not assume that this action, in and of itself, can be used as a fast bzero-type function.
31	Implementation Dependent	Unassigned by the Architecture - available for implementation-dependent use.

Restrictions:

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Bus Error, Cache Error, Address Error, Reserved Instruction, Coprocessor Usable

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:

Prefetch cannot move data to or from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. A prefetch may be used using an address pointer before the validity of the pointer is determined without worrying about an addressing exception.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREFE instruction. Typically, this only occurs in systems which have high-reliability requirements.

Prefetch operations have no effect on cache lines that were previously locked with the CACHE instruction.

Hint field encodings whose function is described as “streamed” or “retained” convey usage intent from software to hardware. Software should not assume that hardware will always prefetch data in an optimal way. If data is to be truly retained, software should use the Cache instruction to lock data into the cache.

31	26 25	21 20	16 15	11 10	6 5	0
COP1X 010011	base	index	hint	0 00000	PREFIX 001111	6

6 5 5 5 5 6

Format: PREFIX hint, index(base)

MIPS64
MIPS32 Release 2

Purpose:

To move data between memory and cache.

Description:

`prefetch_memory[GPR[base] + GPR[index]]`

PREFIX adds the contents of GPR *index* to the contents of GPR *base* to form an effective byte address. The *hint* field supplies information about the way the data is expected to be used.

The only functional difference between the PREF and PREFIX instructions is the addressing mode implemented by the two. Refer to the [PREF](#) instruction for all other details, including the encoding of the *hint* field.

Restrictions:

Operation:

```
vAddr ← GPR[base] + GPR[index]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, Bus Error, Cache Error

Programming Notes:

The PREFIX instruction is only available on processors that implement floating point and should never be generated by compilers in situations other than those in which the corresponding load and store indexed floating point instructions are generated.

Also refer to the corresponding section in the [PREF](#) instruction description.

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt 10110	ft	fs	fd	PUL 101110	6

Format: PUL.PS fd, fs, ft

MIPS64, MIPS32 Release 2

Purpose: Pair Upper Lower

To merge a pair of paired single values with realignment

Description: $FPR[fd] \leftarrow \text{upper}(FPR[fs]) \mid\mid \text{lower}(FPR[ft])$

A new paired-single value is formed by catenating the upper single of FPR *fs* (bits 63..32) and the lower single of FPR *ft* (bits 31..0).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *PS*. If they are not valid, the result is **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

`StoreFPR(fd, PS, ValueFPR(fs, PS)63..32 || ValueFPR(ft, PS)31..0)`

Exceptions:

Coprocessor Unusable, Reserved Instruction

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt 10110	ft	fs	fd	PUU 101111	6

Format: PUU.PS fd, fs, ft

MIPS64, MIPS32 Release 2

Purpose: Pair Upper Upper

To merge a pair of paired single values with realignment

Description: $FPR[fd] \leftarrow \text{upper}(FPR[fs]) \mid\mid \text{upper}(FPR[ft])$

A new paired-single value is formed by catenating the upper single of FPR *fs* (bits 63..32) and the upper single of FPR *ft* (bits 63..32).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *PS*. If they are not valid, the result is **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

`StoreFPR(fd, PS, ValueFPR(fs, PS)63..32 || ValueFPR(ft, PS)63..32)`

Exceptions:

Coprocessor Unusable, Reserved Instruction

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL3 0111 11	0 00 000	rt	rd	0 000 00	RDHWR 11 1011	6

Format: RDHWR rt,rd

MIPS32 Release 2

Purpose: Read Hardware Register

To move the contents of a hardware register to a general purpose register (GPR) if that operation is enabled by privileged software.

The purpose of this instruction is to give user mode access to specific information that is otherwise only visible in kernel mode.

Description: GPR[rt] \leftarrow HWR[rd]

If access is allowed to the specified hardware register, the contents of the register specified by rd is sign-extended and loaded into general register rt. Access control for each register is selected by the bits in the coprocessor 0 *HWREna* register.

The available hardware registers, and the encoding of the rd field for each, are shown in [Table 3.35](#).

Table 3.35 RDHWR Register Numbers

Register Number (rd Value)	Mnemonic	Description										
0	CPUNum	Number of the CPU on which the program is currently running. This register provides read access to the coprocessor 0 <i>EBase_{CPUNum}</i> field.										
1	SYNCI_Step	Address step size to be used with the SYNCI instruction, or zero if no caches need be synchronized. See that instruction's description for the use of this value.										
2	CC	High-resolution cycle counter. This register provides read access to the coprocessor 0 <i>Count</i> Register.										
3	CCRes	Resolution of the CC register. This value denotes the number of cycles between update of the register. For example: <table border="1" style="margin-left: 20px;"> <tr> <th>CCRes Value</th> <th>Meaning</th> </tr> <tr> <td>1</td> <td>CC register increments every CPU cycle</td> </tr> <tr> <td>2</td> <td>CC register increments every second CPU cycle</td> </tr> <tr> <td>3</td> <td>CC register increments every third CPU cycle</td> </tr> <tr> <td>etc.</td> <td></td> </tr> </table>	CCRes Value	Meaning	1	CC register increments every CPU cycle	2	CC register increments every second CPU cycle	3	CC register increments every third CPU cycle	etc.	
CCRes Value	Meaning											
1	CC register increments every CPU cycle											
2	CC register increments every second CPU cycle											
3	CC register increments every third CPU cycle											
etc.												
4-28		These registers numbers are reserved for future architecture use. Access results in a Reserved Instruction Exception.										
29	ULR	User Local Register. This register provides read access to the coprocessor 0 <i>UserLocal</i> register, if it is implemented. In some operating environments, the <i>UserLocal</i> register is a pointer to a thread-specific storage block.										

Table 3.35 RDHWR Register Numbers

Register Number (rd Value)	Mnemonic	Description
30-31		These register numbers are reserved for implementation-dependent use. If they are not implemented, access results in a Reserved Instruction Exception.

Restrictions:

In implementations of Release 1 of the Architecture, this instruction resulted in a Reserved Instruction Exception.

Access to the specified hardware register is enabled if Coprocessor 0 is enabled, or if the corresponding bit is set in the *HWEEna* register. If access is not allowed or the register is not implemented, a Reserved Instruction Exception is signaled.

Operation:

```

case rd
    0: temp ← sign_extend(EBaseCPUNum)
    1: temp ← sign_extend(SYNCI_StepSize())
    2: temp ← sign_extend(Count)
    3: temp ← sign_extend(CountResolution())
    29: temp ← sign_extend_if_32bit_op(UserLocal)
    30: temp ← sign_extend_if_32bit_op(Implementation-Dependent-Value)
    31: temp ← sign_extend_if_32bit_op(Implementation-Dependent-Value)
        otherwise: SignalException(ReservedInstruction)
endcase
GPR[rt] ← temp

function sign_extend_if_32bit_op(value)
    if (width(value) = 64) and Are64bitOperationsEnabled() then
        sign_extend_if_32bit_op ← value
    else
        sign_extend_if_32bit_op ← sign_extend(value)
    endif
end sign_extend_if_32bit_op

```

Exceptions:

Reserved Instruction

31	26 25	21 20	16 15	11 10	0
COP0 0100 00	RDPGPR 01 010	rt	rd	0 000 0000 0000	11

6 5 5 5 11

Format: RDPGPR rd, rt**MIPS32 Release 2****Purpose:** Read GPR from Previous Shadow Set

To move the contents of a GPR from the previous shadow set to a current GPR.

Description: $\text{GPR}[\text{rd}] \leftarrow \text{SGPR}[\text{SRSCtl}_{\text{PSS}}, \text{rt}]$ The contents of the shadow GPR register specified by $\text{SRSCtl}_{\text{PSS}}$ (signifying the previous shadow set number) and rt (specifying the register number within that set) is moved to the current GPR rd .**Restrictions:**

In implementations prior to Release 2 of the Architecture, this instruction resulted in a Reserved Instruction Exception.

Operation: $\text{GPR}[\text{rd}] \leftarrow \text{SGPR}[\text{SRSCtl}_{\text{PSS}}, \text{rt}]$ **Exceptions:**

Coprocessor Unusable

Reserved Instruction

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	0 00000	fs	fd	RECIP 010101	

6 5 5 5 5 6

Format: RECIP.fmtRECIP.S fd, fs
RECIP.D fd, fsMIPS64, MIPS32 Release 2
MIPS64, MIPS32 Release 2**Purpose:** Reciprocal Approximation

To approximate the reciprocal of an FP value (quickly)

Description: FPR[fd] \leftarrow 1.0 / FPR[fs]The reciprocal of the value in FPR *fs* is approximated and placed into FPR *fd*. The operand and result are values in format *fmt*.

The numeric accuracy of this operation is implementation dependent; it does not meet the accuracy specified by the IEEE 754 Floating Point standard. The computed result differs from the both the exact result and the IEEE-mandated representation of the exact result by no more than one unit in the least-significant place (ULP).

It is implementation dependent whether the result is affected by the current rounding mode in *FCSR*.**Restrictions:**The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.**Operation:**

StoreFPR(fd, fmt, 1.0 / valueFPR(fs, fmt))

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Division-by-zero, Unimplemented Op, Invalid Op, Overflow, Underflow

31	26 25	22 21 20	16 15	11 10	6 5	0
SPECIAL 000000	0000	R 1	rt	rd	sa	SRL 000010

6 4 1 5 5 5 6

Format: ROTR rd, rt, sa**SmartMIPS Crypto, MIPS32 Release 2****Purpose:** Rotate Word Right

To execute a logical right-rotate of a word by a fixed number of bits

Description: $GPR[rd] \leftarrow GPR[rt] \leftrightarrow(\text{right}) sa$ The contents of the low-order 32-bit word of GPR *rt* are rotated right; the word result is sign-extended and placed in GPR *rd*. The bit-rotate amount is specified by *sa*.**Restrictions:**If GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.**Operation:**

```

if NotWordValue(GPR[rt]) or
    ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
    UNPREDICTABLE
endif
s ← sa
temp ← GPR[rt]s-1..0 || GPR[rt]31..s
GPR[rd] ← sign_extend(temp)

```

Exceptions:

Reserved Instruction

31	26 25	21 20	16 15	11 10	7 6 5	0
SPECIAL 000000	rs	rt	rd	0000	R 1	SRLV 000110

6 5 5 5 4 1 6

Format: ROTRV rd, rt, rs**SmartMIPS Crypto, MIPS32 Release 2****Purpose:** Rotate Word Right Variable

To execute a logical right-rotate of a word by a variable number of bits

Description: GPR[rd] \leftarrow GPR[rt] \leftrightarrow (right) GPR[rs]

The contents of the low-order 32-bit word of GPR rt are rotated right; the word result is sign-extended and placed in GPR rd. The bit-rotate amount is specified by the low-order 5 bits of GPR rs.

Restrictions:If GPR rt does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.**Operation:**

```

if NotWordValue(GPR[rt]) or
    ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
    UNPREDICTABLE
endif
s  $\leftarrow$  GPR[rs]4..0
temp  $\leftarrow$  GPR[rt]s-1..0 || GPR[rt]31..s
GPR[rd]  $\leftarrow$  sign_extend(temp)

```

Exceptions:

Reserved Instruction

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	0 00000	fs	fd	ROUND.L 001000	

6 5 5 5 5 6

Format: ROUND.L fmtROUND.L.S fd, fs
ROUND.L.D fd, fsMIPS64, MIPS32 Release 2
MIPS64, MIPS32 Release 2**Purpose:** Floating Point Round to Long Fixed Point

To convert an FP value to 64-bit fixed point, rounding to nearest

Description: FPR[fd] \leftarrow convert_and_round(FPR[fs])The value in FPR *fs*, in format *fmt*, is converted to a value in 64-bit long fixed point format and rounded to nearest/even (rounding mode 0). The result is placed in FPR *fd*.When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}-1$, is written to *fd*.**Restrictions:**The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed point; if they are not valid, the result is **UNPREDICTABLE**.The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.**Operation:**

StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	0 00000	fs	fd	ROUND.W 001100	

6 5 5 5 5 6

Format: ROUND.W fmtROUND.W.S fd, fs
ROUND.W.D fd, fsMIPS32
MIPS32**Purpose:** Floating Point Round to Word Fixed Point

To convert an FP value to 32-bit fixed point, rounding to nearest

Description: FPR[fd] \leftarrow convert_and_round(FPR[fs])The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format rounding to nearest/even (rounding mode 0). The result is placed in FPR *fd*.When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *fd*.**Restrictions:**The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed point; if they are not valid, the result is **UNPREDICTABLE**.The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.**Operation:**

StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	0 00000	fs	fd	RSQRT 010110	

6 5 5 5 5 6

Format: RSQRT.fmtRSQRT.S fd, fs
RSQRT.D fd, fsMIPS64, MIPS32 Release 2
MIPS64, MIPS32 Release 2**Purpose:** Reciprocal Square Root Approximation

To approximate the reciprocal of the square root of an FP value (quickly)

Description: $FPR[fd] \leftarrow 1.0 / \sqrt{FPR[fs]}$ The reciprocal of the positive square root of the value in FPR *fs* is approximated and placed into FPR *fd*. The operand and result are values in format *fmt*.

The numeric accuracy of this operation is implementation dependent; it does not meet the accuracy specified by the IEEE 754 Floating Point standard. The computed result differs from both the exact result and the IEEE-mandated representation of the exact result by no more than two units in the least-significant place (ULP).

The effect of the current FCSR rounding mode on the result is implementation dependent.

Restrictions:The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.**Operation:**

StoreFPR(fd, fmt, 1.0 / SquareRoot(valueFPR(fs, fmt)))

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Division-by-zero, Unimplemented Operation, Invalid Operation, Overflow, Underflow

31	26 25	21 20	16 15	0
SB 101000	base	rt	offset	

6 5 5 16

Format: SB rt, offset(base)**MIPS32****Purpose:** Store Byte

To store a byte to memory

Description: $\text{memory}[\text{GPR}[base] + \text{offset}] \leftarrow \text{GPR}[rt]$ The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.**Restrictions:**

None

Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
bytesel ← vAddr2..0 xor BigEndianCPU3
datadoubleword ← GPR[rt]63-8*bytesel..0 || 08*bytesel
StoreMemory (CCA, BYTE, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch

31	26 25	21 20	16 15		7 6 5	0
SPECIAL3 011111	base	rt	offset	0	SBE 011100	6

6 5 5 9 1 6

Format: SBE rt, offset(base)**MIPS32****Purpose:** Store Byte EVA

To store a byte to user mode virtual address space when executing in kernel mode.

Description: memory[GPR[base] + offset] \leftarrow GPR[rt]

The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The SBE instruction functions in exactly the same fashion as the SB instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the *Config5_{EVA}* field being set to one.

Restrictions:

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Operation:

```
vAddr  $\leftarrow$  sign_extend(offset) + GPR[base]
(pAddr, CCA)  $\leftarrow$  AddressTranslation (vAddr, DATA, STORE)
pAddr  $\leftarrow$  pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
bytesel  $\leftarrow$  vAddr2..0 xor BigEndianCPU3
datadoubleword  $\leftarrow$  GPR[rt]63-8*bytesel..0 || 08*bytesel
StoreMemory (CCA, BYTE, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill

TLB Invalid

Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable

31	26 25	21 20	16 15	0
SC 111000	base	rt	offset	

6 5 5 16

Format: SC rt, offset(base)**MIPS32****Purpose:** Store Conditional Word

To store a word to memory to complete an atomic read-modify-write

Description: if atomic_update then memory[GPR[base] + offset] \leftarrow GPR[rt], GPR[rt] \leftarrow 1
else GPR[rt] \leftarrow 0

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The least-significant 32-bit word in GPR *rt* is conditionally stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The SC completes the RMW sequence begun by the preceding LL instruction executed on the processor. To complete the RMW sequence atomically, the following occur:

- The least-significant 32-bit word of GPR *rt* is stored into memory at the location specified by the aligned effective address.
- A 1, indicating success, is written into GPR *rt*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *rt*.

If either of the following events occurs between the execution of LL and SC, the SC fails:

- A coherent store is completed by another processor or coherent I/O module into the block of synchronizable physical memory containing the word. The size and alignment of the block is implementation dependent, but it is at least one word and at most the minimum page size.
- An ERET instruction is executed.

If either of the following events occurs between the execution of LL and SC, the SC may succeed or it may fail; the success or failure is not predictable. Portable programs should not cause one of these events.

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LL/SC.
- The instructions executed starting with the LL and ending with the SC do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

The following conditions must be true or the result of the SC is **UNPREDICTABLE**:

- Execution of SC must have been preceded by execution of an LL instruction.
- An RMW sequence executed without intervening events that would cause the SC to fail must use the same address in the LL and SC. The address is the same if the virtual address, physical address, and cacheability & coherency attribute are identical.

Atomic RMW is provided only for synchronizable memory locations. A synchronizable memory location is one that is associated with the state and logic necessary to implement the LL/SC semantics. Whether a memory location is

synchronizable depends on the processor and system configurations, and on the memory access type used for the location:

- **Uniprocessor atomicity:** To provide atomic RMW on a single processor, all accesses to the location must be made with memory access type of either *cached noncoherent* or *cached coherent*. All accesses must be to one or the other access type, and they may not be mixed.
- **MP atomicity:** To provide atomic RMW among multiple processors, all accesses to the location must be made with a memory access type of *cached coherent*.
- **I/O System:** To provide atomic RMW with a coherent I/O system, all accesses to the location must be made with a memory access type of *cached coherent*. If the I/O system does not use coherent memory operations, then atomic RMW cannot be provided with respect to the I/O reads and writes.

Restrictions:

The addressed location must have a memory access type of *cached noncoherent* or *cached coherent*; if it does not, the result is **UNPREDICTABLE**.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
datadoubleword ← GPR[rt]63-8*bytesel..0 || 08*bytesel
if LLbit then
    StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 063 || 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	21 20	16 15	7 6 5	0
SPECIAL3 011111	base	rt	offset	0	SCE 011110

6 5 5 9 1 6

Format: SCE rt, offset(base)

MIPS32

Purpose: Store Conditional Word EVA

To store a word to user mode virtual memory while operating in kernel mode to complete an atomic read-modify-write

Description: if atomic_update then memory[GPR[base] + offset] \leftarrow GPR[rt], GPR[rt] \leftarrow 1
else GPR[rt] \leftarrow 0

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The least-significant 32-bit word in GPR *rt* is conditionally stored in memory at the location specified by the aligned effective address. The 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The SCE completes the RMW sequence begun by the preceding LLE instruction executed on the processor. To complete the RMW sequence atomically, the following occur:

- The least-significant 32-bit word of GPR *rt* is stored 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 LLE and SCE, the SCE may succeed or it may fail; the success or failure is not predictable. Portable programs should not cause one of these events.

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LLE/SCE.
- The instructions executed starting with the LLE and ending with the SCE do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

The following conditions must be true or the result of the SCE is **UNPREDICTABLE**:

- Execution of SCE must have been preceded by execution of an LLE instruction.
- An RMW sequence executed without intervening events that would cause the SCE to fail must use the same address in the LLE and SCE. The address is the same if the virtual address, physical address, and cacheability & coherency attribute are identical.

Atomic RMW is provided only for synchronizable memory locations. A synchronizable memory location is one that

is associated with the state and logic necessary to implement the LLE/SCE semantics. Whether a memory location is synchronizable depends on the processor and system configurations, and on the memory access type used for the location:

- **Uniprocessor atomicity:** To provide atomic RMW on a single processor, all accesses to the location must be made with memory access type of either *cached noncoherent* or *cached coherent*. All accesses must be to one or the other access type, and they may not be mixed.
- **MP atomicity:** To provide atomic RMW among multiple processors, all accesses to the location must be made with a memory access type of *cached coherent*.
- **I/O System:** To provide atomic RMW with a coherent I/O system, all accesses to the location must be made with a memory access type of *cached coherent*. If the I/O system does not use coherent memory operations, then atomic RMW cannot be provided with respect to the I/O reads and writes.

The SCE instruction functions in exactly the same fashion as the SC instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the *Config5_{EVA}* field being set to one.

Restrictions:

The addressed location must have a memory access type of *cached noncoherent* or *cached coherent*; if it does not, the result is **UNPREDICTABLE**.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
datadoubleword ← GPR[rt]63-8*bytesel..0 || 08*bytesel
if LLbit then
    StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 063 || LLbit
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch, Reserved Instruction, Coprocessor Unusable

Programming Notes:

LLE and SCE are used to atomically update memory locations, as shown below.

```
L1:
    LLE    T1, (T0)  # load counter
    ADDI   T2, T1, 1 # increment
    SCE    T2, (T0)  # try to store, checking for atomicity
```

```
BEQ    T2, 0, L1 # if not atomic (0), try again  
NOP          # branch-delay slot
```

Exceptions between the LLE and SCE cause SCE to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LLE and SCE function on a single processor for *cached noncoherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.

31	26 25	21 20	16 15	0
SCD 111100	base	rt	offset	

6 5 5 16

Format: SCD rt, offset(base)**MIPS64****Purpose:** Store Conditional Doubleword

To store a doubleword to memory to complete an atomic read-modify-write

Description: if atomic_update then memory[GPR[base] + offset] ← GPR[rt], GPR[rt] ← 1
else GPR[rt] ← 0

The LLD and SCD instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The 64-bit doubleword in GPR *rt* is conditionally stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The SCD completes the RMW sequence begun by the preceding LLD instruction executed on the processor. If it would complete the RMW sequence atomically, the following occur:

- The 64-bit doubleword of GPR *rt* is stored into memory at the location specified by the aligned effective address.
- A 1, indicating success, is written into GPR *rt*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *rt*.

If either of the following events occurs between the execution of LLD and SCD, the SCD fails:

- A coherent store is completed by another processor or coherent I/O module into the block of synchronizable physical memory containing the doubleword. The size and alignment of the block is implementation dependent, but it is at least one doubleword and at most the minimum page size.
- An ERET instruction is executed.

If either of the following events occurs between the execution of LLD and SCD, the SCD may succeed or it may fail; success or failure is not predictable. Portable programs should not cause these events:

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LLD/SCD.
- The instructions executed starting with the LLD and ending with the SCD do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

The following two conditions must be true or the result of the SCD is **UNPREDICTABLE**:

- Execution of the SCD must be preceded by execution of an LLD instruction.
- An RMW sequence executed without intervening events that would cause the SCD to fail must use the same address in the LLD and SCD. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.

Atomic RMW is provided only for synchronizable memory locations. A synchronizable memory location is one that is associated with the state and logic necessary to implement the LL/SC semantics. Whether a memory location is synchronizable depends on the processor and system configurations, and on the memory access type used for the

location:

- **Uniprocessor atomicity:** To provide atomic RMW on a single processor, all accesses to the location must be made with memory access type of either *cached noncoherent* or *cached coherent*. All accesses must be to one or the other access type, and they may not be mixed.
- **MP atomicity:** To provide atomic RMW among multiple processors, all accesses to the location must be made with a memory access type of *cached coherent*.
- **I/O System:** To provide atomic RMW with a coherent I/O system, all accesses to the location must be made with a memory access type of *cached coherent*. If the I/O system does not use coherent memory operations, then atomic RMW cannot be provided with respect to the I/O reads and writes.

Restrictions:

The addressed location must have a memory access type of *cached noncoherent* or *cached coherent*; if it does not, the result is **UNPREDICTABLE**.

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
datadoubleword ← GPR[rt]
if LLbit then
    StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 063 || LLbit
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Reserved Instruction, Watch

Programming Notes:

LLD and SCD are used to atomically update memory locations, as shown below.

```
L1:
    LLD    T1, (T0)  # load counter
    ADDI   T2, T1, 1 # increment
    SCD    T2, (T0)  # try to store,
            # checking for atomicity
    BEQ    T2, 0, L1 # if not atomic (0), try again
    NOP
```

Exceptions between the LLD and SCD cause SCD to fail, so persistent exceptions must be avoided. Some examples of such exceptions are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LLD and SCD function on a single processor for *cached noncoherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.

31	26 25	21 20	16 15	0
SD 111111	base	rt	offset	

6 5 5 16

Format: SD rt, offset(base)**MIPS64****Purpose:** Store Doubleword

To store a doubleword to memory

Description: memory[GPR[base] + offset] \leftarrow GPR[rt]The 64-bit doubleword in GPR *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.**Restrictions:**

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

Operation:

```

vAddr  $\leftarrow$  sign_extend(offset) + GPR[base]
if vAddr2..0  $\neq$  03 then
    SignalException(AddressError)
endif
(pAddr, CCA)  $\leftarrow$  AddressTranslation (vAddr, DATA, STORE)
datadoubleword  $\leftarrow$  GPR[rt]
StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Reserved Instruction, Watch

31	26 25		6 5	0
SPECIAL2 011100 6		code - use syscall 20	SDBBP 111111 6	

Format: SDBBP code**EJTAG****Purpose:** Software Debug Breakpoint

To cause a debug breakpoint exception

Description:

This instruction causes a debug exception, passing control to the debug exception handler. If the processor is executing in Debug Mode when the SDBBP instruction is executed, the exception is a Debug Mode Exception, which sets the DebugDExcCode field to the value 0x9 (Bp). The code field can be used for passing information to the debug exception handler, and is retrieved by the debug exception handler only by loading the contents of the memory word containing the instruction, using the DEPC register. The CODE field is not used in any way by the hardware.

Restrictions:**Operation:**

```

If DebugDM = 0 then
    SignalDebugBreakpointException()
else
    SignalDebugModeBreakpointException()
endif

```

Exceptions:

Debug Breakpoint Exception

Debug Mode Breakpoint Exception

31	26 25	21 20	16 15	0
SDC1 111101	base	ft	offset	

6 5 5 16

Format: SDC1 ft, offset(base)**MIPS32****Purpose:** Store Doubleword from Floating Point

To store a doubleword from an FPR to memory

Description: $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{FPR}[\text{ft}]$ 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 EffectiveAddress_{2..0} ≠ 0 (not doubleword-aligned).**Operation:**

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

31	26 25	21 20	16 15	0
SDC2 111110	base	rt	offset	

6 5 5 16

Format: SDC2 rt, offset(base)**MIPS32****Purpose:** Store Doubleword from Coprocessor 2

To store a doubleword from a Coprocessor 2 register to memory

Description: $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{CPR}[2, \text{rt}, 0]$ The 64-bit doubleword in Coprocessor 2 register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.**Restrictions:**An Address Error exception occurs if $\text{EffectiveAddress}_{2..0} \neq 0$ (not doubleword-aligned).**Operation:**

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← CPR[2, rt, 0]
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

31	26 25	21 20	16 15	0
SDL 101100	base	rt	offset	
6	5	5	16	

Format: `SDL rt, offset(base)`**MIPS64****Purpose:** Store Doubleword Left

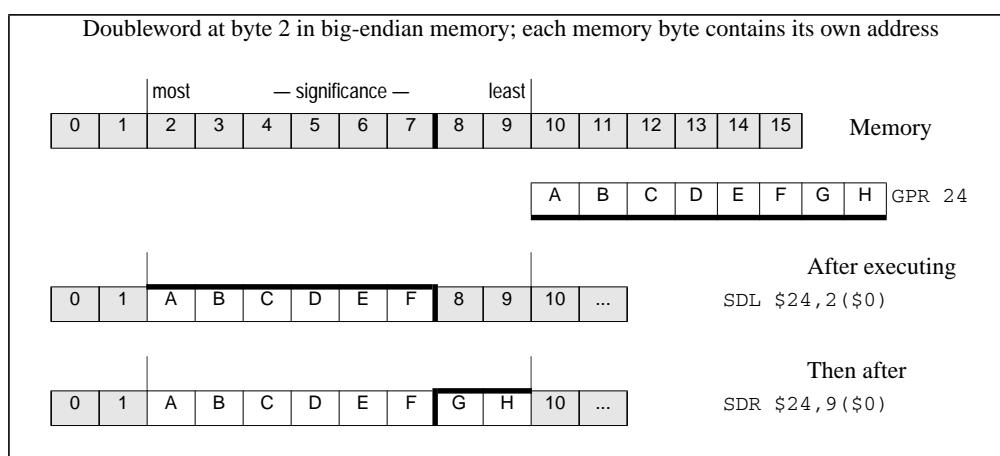
To store the most-significant part of a doubleword to an unaligned memory address

Description: `memory[GPR[base] + offset] ← Some_Bytes_From GPR[rt]`

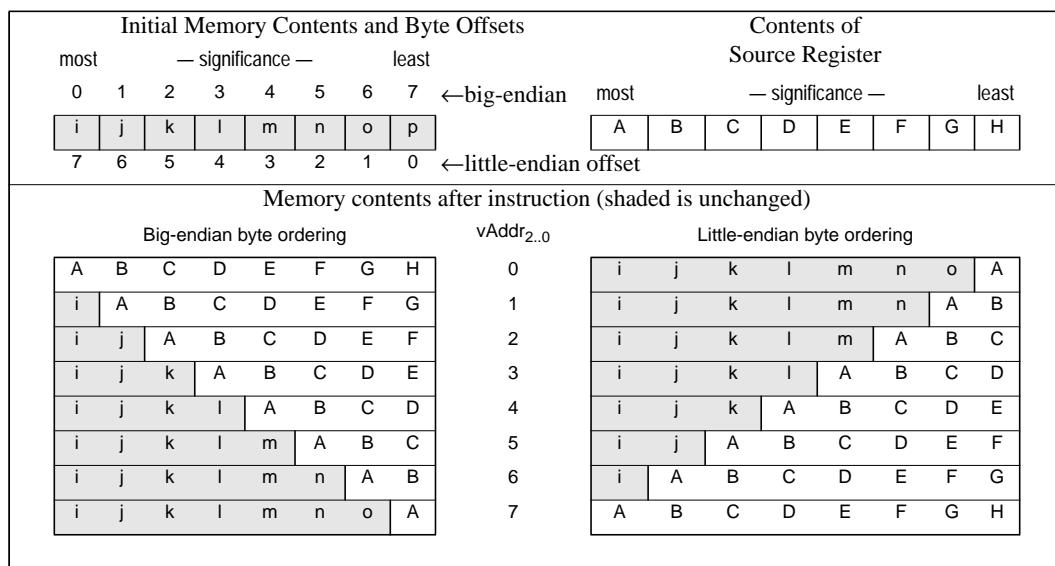
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 8 consecutive bytes forming a doubleword (*DW*) in memory, starting at an arbitrary byte boundary.

A part of *DW*, the most-significant 1 to 8 bytes, is in the aligned doubleword containing *EffAddr*. The same number of most-significant (left) bytes of GPR *rt* are stored into these bytes of *DW*.

The figure below illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 6 bytes, is located in the aligned doubleword containing the most-significant byte at 2. First, SDL stores the 6 most-significant bytes of the source register into these bytes of *DW*. Next, the complementary SDR instruction stores the remainder of *DW*.

Figure 3.24 Unaligned Doubleword Store With SDL and SDR

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned doubleword—that is, the low 3 bits of the address ($vAddr_{2..0}$)—and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes stored for every combination of offset and byte ordering.

Figure 3.25 Bytes Stored by an SDL Instruction**Restrictions:****Operation:**

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
If BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..3 || 03
endif
bytesel ← vAddr2..0 xor BigEndianCPU3
datadoubleword ← 056-8*bytesel || GPR[rt]63..56-8*bytesel
StoreMemory (CCA, byte, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Reserved Instruction, Watch

31	26 25	21 20	16 15	0
SDR 101101	base	rt	offset	

6 5 5 16

Format: SDR rt, offset(base)**MIPS64****Purpose:** Store Doubleword Right

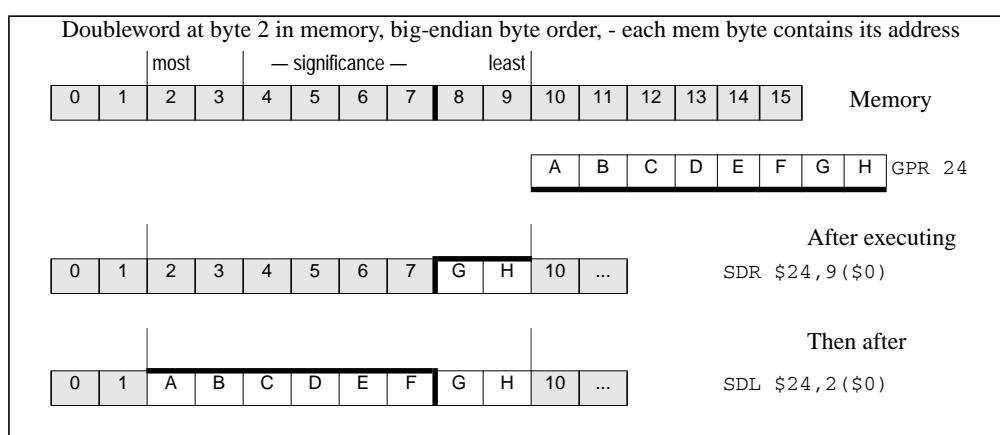
To store the least-significant part of a doubleword to an unaligned memory address

Description: $\text{memory}[\text{GPR}[base] + \text{offset}] \leftarrow \text{Some_Bytes_From GPR}[rt]$

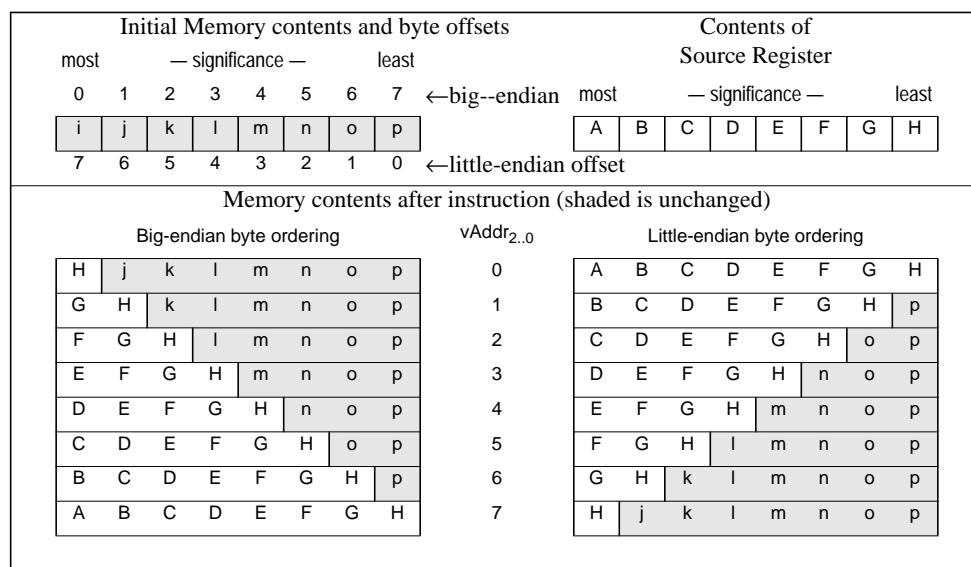
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of 8 consecutive bytes forming a doubleword (*DW*) in memory, starting at an arbitrary byte boundary.

A part of *DW*, the least-significant 1 to 8 bytes, is in the aligned doubleword containing *EffAddr*. The same number of least-significant (right) bytes of GPR *rt* are stored into these bytes of *DW*.

Figure 3-25 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 2 bytes, is located in the aligned doubleword containing the least-significant byte at 9. First, SDR stores the 2 least-significant bytes of the source register into these bytes of *DW*. Next, the complementary SDL stores the remainder of *DW*.

Figure 3.26 Unaligned Doubleword Store With SDR and SDL

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned doubleword—that is, the low 3 bits of the address ($v\text{Addr}_{2..0}$)—and the current byte ordering mode of the processor (big- or little-endian). Figure 3-26 shows the bytes stored for every combination of offset and byte-ordering.

Figure 3.27 Bytes Stored by an SDR Instruction**Restrictions:****Operation:**

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
If BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..3 || 03
endif
bytesel ← vAddr1..0 xor BigEndianCPU3
datadoubleword ← GPR[rt]63-8*bytesel || 08*bytesel
StoreMemory (CCA, DOUBLEWORD-byte, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Reserved Instruction, Watch

31	26 25	21 20	16 15	11 10	6 5	0
COP1X 010011	base	index	fs	0 00000	SDXC1 001001	6

Format: SDXC1 fs, index(base)

MIPS64
MIPS32 Release 2

Purpose: Store Doubleword Indexed from Floating Point

To store a doubleword from an FPR to memory (GPR+GPR addressing)

Description: memory[GPR[base] + GPR[index]] \leftarrow FPR[fs]

The 64-bit doubleword in FPR_{fs} is stored in memory at the location specified by the aligned effective address. The contents of GPR *index* and GPR *base* are added to form the effective address.

Restrictions:

An Address Error exception occurs if EffectiveAddress_{2..0} \neq 0 (not doubleword-aligned).

Operation:

```
vAddr  $\leftarrow$  GPR[base] + GPR[index]
if vAddr2..0  $\neq$  03 then
    SignalException(AddressError)
endif
(pAddr, CCA)  $\leftarrow$  AddressTranslation(vAddr, DATA, STORE)
datadoubleword  $\leftarrow$  ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Coprocessor Unusable, Address Error, Reserved Instruction, Watch.

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL3 011111	0 00000	rt	rd	SEB 10000	BSHFL 100000	6

Format: SEB rd, rt**MIPS32 Release 2****Purpose:** Sign-Extend Byte

To sign-extend the least significant byte of GPR *rt* and store the value into GPR *rd*.

Description: $\text{GPR}[\text{rd}] \leftarrow \text{SignExtend}(\text{GPR}[\text{rt}]_{7..0})$

The least significant byte from GPR *rt* is sign-extended and stored in GPR *rd*.

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

If GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
GPR[rd] ← sign_extend(GPR[rt]7..0)
```

Exceptions:

Reserved Instruction

Programming Notes:

For symmetry with the SEB and SEH instructions, one would expect that there would be ZEB and ZEH instructions that zero-extend the source operand. Similarly, one would expect that the SEW and ZEW instructions would exist to sign- or zero-extend a word to a doubleword. These instructions do not exist because there are functionally-equivalent instructions already in the instruction set. The following table shows the instructions providing the equivalent functions.

Expected Instruction	Function	Equivalent Instruction
ZEB rx, ry	Zero-Extend Byte	ANDI rx, ry, 0xFF
ZEH rx, ry	Zero-Extend Halfword	ANDI rx, ry, 0xFFFF
SEW rx, ry	Sign-Extend Word	SLL rx, ry, 0
ZEW rx, rx ¹	Zero-Extend Word	DINSP32 rx, r0, 32, 32

1. The equivalent instruction uses rx for both source and destination, so the expected instruction is limited to one register

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL3 011111	0 00000	rt	rd	SEH 11000	BSHFL 100000	6

Format: SEH rd, rt**MIPS32 Release 2****Purpose:** Sign-Extend HalfwordTo sign-extend the least significant halfword of GPR *rt* and store the value into GPR *rd*.**Description:** $\text{GPR}[rd] \leftarrow \text{SignExtend}(\text{GPR}[rt]_{15..0})$ The least significant halfword from GPR *rt* is sign-extended and stored in GPR *rd*.**Restrictions:**

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

If GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.**Operation:**

```

if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
GPR[rd] ← sign_extend(GPR[rt]15..0)

```

Exceptions:

Reserved Instruction

Programming Notes:

The SEH instruction can be used to convert two contiguous halfwords to sign-extended word values in three instructions. For example:

```

lw      t0, 0(a1)          /* Read two contiguous halfwords */
seh    t1, t0              /* t1 = lower halfword sign-extended to word */
sra    t0, t0, 16           /* t0 = upper halfword sign-extended to word */

```

Zero-extended halfwords can be created by changing the SEH and SRA instructions to ANDI and SRL instructions, respectively.

For symmetry with the SEB and SEH instructions, one would expect that there would be ZEB and ZEH instructions that zero-extend the source operand. Similarly, one would expect that the SEW and ZEW instructions would exist to sign- or zero-extend a word to a doubleword. These instructions do not exist because there are functionally-equivalent instructions already in the instruction set. The following table shows the instructions providing the equivalent functions.

Expected Instruction	Function	Equivalent Instruction
ZEB rx, ry	Zero-Extend Byte	ANDI rx, ry, 0xFF
ZEH rx, ry	Zero-Extend Halfword	ANDI rx, ry, 0xFFFF
SEW rx, ry	Sign-Extend Word	SLL rx, ry, 0
ZEW rx, rx ¹	Zero-Extend Word	DINSP32 rx, r0, 32, 32

1. The equivalent instruction uses rx for both source and destination, so the expected instruction is limited to one register

31	26 25	21 20	16 15	0
SH 101001	base	rt		offset

6 5 5 16

Format: SH rt, offset(base)**MIPS32****Purpose:** Store Halfword

To store a halfword to memory

Description: memory[GPR[base] + offset] \leftarrow GPR[rt]The least-significant 16-bit halfword of register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.**Restrictions:**

The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation:

```

vAddr  $\leftarrow$  sign_extend(offset) + GPR[base]
if vAddr0  $\neq$  0 then
    SignalException(AddressError)
endif
(pAddr, CCA)  $\leftarrow$  AddressTranslation(vAddr, DATA, STORE)
pAddr  $\leftarrow$  pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian2 || 0))
bytesel  $\leftarrow$  vAddr2..0 xor (BigEndianCPU2 || 0)
datadoubleword  $\leftarrow$  GPR[rt]63-8*bytesel..0 || 08*bytesel
StoreMemory(CCA, HALFWORD, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

31	26 25	21 20	16 15		7 6 5	0
SPECIAL3 011111	base	rt		offset	0	SHE 011101

6 5 5 9 1 6

Format: SHE rt, offset(base)

MIPS32

Purpose: Store Halfword EVA

To store a halfword to user mode virtual address space when executing in kernel mode.

Description: memory[GPR[base] + offset] \leftarrow GPR[rt]

The least-significant 16-bit halfword of register *rt* is stored in memory at the location specified by the aligned effective address. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The SHE instruction functions in exactly the same fashion as the SH instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the *Config5_{EVA}* field being set to one.

Restrictions:

Only usable in kernel mode when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

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  $\leftarrow$  sign_extend(offset) + GPR[base]
(pAddr, CCA)  $\leftarrow$  AddressTranslation (vAddr, DATA, STORE)
pAddr  $\leftarrow$  pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian2 || 0))
bytesel  $\leftarrow$  vAddr2..0 xor (BigEndianCPU2 || 0)
datadoubleword  $\leftarrow$  GPR[rt]63-8*bytesel..0 || 08*bytesel
StoreMemory (CCA, HALFWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill

TLB Invalid

Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	0 00000	rt	rd	sa	SLL 000000	6

Format: SLL rd, rt, sa**MIPS32****Purpose:** Shift Word Left Logical

To left-shift a word by a fixed number of bits

Description: GPR[rd] \leftarrow GPR[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 sign-extended and placed in GPR rd. The bit-shift amount is specified by sa.

Restrictions:

None

Operation:

```

s  $\leftarrow$  sa
temp  $\leftarrow$  GPR[rt](31-s)... || 0s
GPR[rd]  $\leftarrow$  sign_extend(temp)

```

Exceptions:

None

Programming Notes:

Unlike nearly all other word operations, the SLL input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign-extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign-extends it.

SLL r0, r0, 0, expressed as NOP, is the assembly idiom used to denote no operation.

SLL r0, r0, 1, expressed as SSNOP, is the assembly idiom used to denote no operation that causes an issue break on superscalar processors.

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	SLLV 000100	

Format: SLLV rd, rt, rs**MIPS32****Purpose:** Shift Word Left Logical Variable

To left-shift a word by a variable number of bits

Description: GPR[rd] \leftarrow GPR[rt] $<<$ rs

The contents of the low-order 32-bit word of GPR rt are shifted left, inserting zeros into the emptied bits; the result word is sign-extended and placed in GPR rd. The bit-shift amount is specified by the low-order 5 bits of GPR rs.

Restrictions:

None

Operation:

```

s  $\leftarrow$  GPR[rs]4..0
temp  $\leftarrow$  GPR[rt](31-s)..0 || 0s
GPR[rd]  $\leftarrow$  sign_extend(temp)

```

Exceptions:

None

Programming Notes:

Unlike nearly all other word operations, the input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign-extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign-extends it.

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	SLT 101010	

6 5 5 5 5 6

Format: SLT rd, rs, rt**MIPS32****Purpose:** Set on Less Than

To record the result of a less-than comparison

Description: $GPR[rd] \leftarrow (GPR[rs] < GPR[rt])$ Compare the contents of GPR *rs* and GPR *rt* as signed integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```

if GPR[rs] < GPR[rt] then
    GPR[rd] ← 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:** Set on Less Than Immediate

To record the result of a less-than comparison with a constant

Description: $GPR[rt] \leftarrow (GPR[rs] < \text{immediate})$ Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```

if GPR[rs] < sign_extend(immediate) then
    GPR[rt] ← 0GPRLEN-1 || 1
else
    GPR[rt] ← 0GPRLEN
endif

```

Exceptions:

None

31	26 25	21 20	16 15	0
SLTIU 001011	rs	rt	immediate	

6 5 5 16

Format: SLTIU rt, rs, immediate**MIPS32****Purpose:** Set on Less Than Immediate Unsigned

To record the result of an unsigned less-than comparison with a constant

Description: $GPR[rt] \leftarrow (GPR[rs] < \text{immediate})$ Compare the contents of GPR *rs* and the sign-extended 16-bit *immediate* as unsigned integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```

if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
    GPR[rt] ← 0GPRLEN-1 || 1
else
    GPR[rt] ← 0GPRLEN
endif

```

Exceptions:

None

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	SLTU 101011	

Format: SLTU rd, rs, rt

MIPS32

Purpose: Set on Less Than Unsigned

To record the result of an unsigned less-than comparison

Description: $GPR[rd] \leftarrow (GPR[rs] < GPR[rt])$

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```

if (0 || GPR[rs]) < (0 || GPR[rt]) then
    GPR[rd] ← 0GPRLEN-1 || 1
else
    GPR[rd] ← 0GPRLEN
endif

```

Exceptions:

None

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	0 00000	fs	fd	SQRT 000100	

Format: SQRT(fmt)
 SQRT.S fd, fs
 SQRT.D fd, fs

MIPS32
 MIPS32

Purpose: Floating Point Square Root

To compute the square root of an FP value

Description: FPR[fd] \leftarrow SQRT(FPR[fs])

The square root of the value in FPR *fs* is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operand and result are values in format *fmt*.

If the value in FPR *fs* corresponds to –0, the result is –0.

Restrictions:

If the value in FPR *fs* is less than 0, an Invalid Operation condition is raised.

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

StoreFPR(fd, fmt, SquareRoot(ValueFPR(fs, fmt)))

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Inexact, Unimplemented Operation

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	0 00000	rt	rd	sa	SRA 000011	

Format: SRA rd, rt, sa**MIPS32****Purpose:** Shift Word Right Arithmetic

To execute an arithmetic right-shift of a word by a fixed number of bits

Description: GPR[rd] \leftarrow GPR[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 sign-extended and placed in GPR *rd*. The bit-shift amount is specified by *sa*.**Restrictions:**On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.**Operation:**

```

if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
s  $\leftarrow$  sa
temp  $\leftarrow$  (GPR[rt]31)s || GPR[rt]31..s
GPR[rd]  $\leftarrow$  sign_extend(temp)

```

Exceptions:

None

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	SRAV 000111	

6 5 5 5 5 6

Format: SRAV rd, rt, rs**MIPS32****Purpose:** Shift Word Right Arithmetic Variable

To execute an arithmetic right-shift of a word by a variable number of bits

Description: GPR[rd] \leftarrow GPR[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 sign-extended and placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```

if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
s ← GPR[rs]4..0
temp ← (GPR[rt]31)s || GPR[rt]31..s
GPR[rd] ← sign_extend(temp)

```

Exceptions:

None

31	26 25	22 21 20	16 15	11 10	6 5	0
SPECIAL 000000	0000	R 0	rt	rd	sa	SRL 000010

6 4 1 5 5 5 6

Format: SRL rd, rt, sa**MIPS32****Purpose:** Shift Word Right Logical

To execute a logical right-shift of a word by a fixed number of bits

Description: GPR[rd] \leftarrow GPR[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 sign-extended and placed in GPR rd. The bit-shift amount is specified by sa.

Restrictions:On 64-bit processors, if GPR rt does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.**Operation:**

```

if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
s  $\leftarrow$  sa
temp  $\leftarrow$  0s || GPR[rt]31..s
GPR[rd]  $\leftarrow$  sign_extend(temp)

```

Exceptions:

None

31	26 25	21 20	16 15	11 10	7 6 5	0
SPECIAL 000000	rs	rt	rd	0000	R 0	SRLV 000110

Format: SRLV rd, rt, rs

MIPS32

Purpose: Shift Word Right Logical Variable

To execute a logical right-shift of a word by a variable number of bits

Description: GPR[rd] \leftarrow GPR[rt] \gg GPR[rs] (logical)

The contents of the low-order 32-bit word of GPR rt are shifted right, inserting zeros into the emptied bits; the word result is sign-extended and placed in GPR rd. The bit-shift amount is specified by the low-order 5 bits of GPR rs.

Restrictions:

On 64-bit processors, if GPR rt does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```

if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
s  $\leftarrow$  GPR[rs]4..0
temp  $\leftarrow$  0s || GPR[rt]31..s
GPR[rd]  $\leftarrow$  sign_extend(temp)

```

Exceptions:

None

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	0 00000	0 00000	0 00000	1 00001	SLL 000000	6

Format: SSNOP**MIPS32****Purpose:** Superscalar No Operation

Break superscalar issue on a superscalar processor.

Description:

SSNOP is the assembly idiom used to denote superscalar no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 1.

This instruction alters the instruction issue behavior on a superscalar processor by forcing the SSNOP instruction to single-issue. The processor must then end the current instruction issue between the instruction previous to the SSNOP and the SSNOP. The SSNOP then issues alone in the next issue slot.

On a single-issue processor, this instruction is a NOP that takes an issue slot.

Restrictions:

None

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:

```
mfc0    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.

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	SUB 100010	6

Format: SUB rd, rs, rt

MIPS32

Purpose: Subtract Word

To subtract 32-bit integers. If overflow occurs, then trap

Description: GPR[rd] \leftarrow GPR[rs] – GPR[rt]

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* to produce a 32-bit result. If the subtraction results in 32-bit 2's complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is sign-extended and placed into GPR *rd*.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```

if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp  $\leftarrow$  (GPR[rs]31 | GPR[rs]31..0) – (GPR[rt]31 | GPR[rt]31..0)
if temp32  $\neq$  temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rd]  $\leftarrow$  sign_extend(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

Format: SUB fmt

SUB.S fd, fs, ft

MIPS32

SUB.D fd, fs, ft

MIPS32

SUB.PS fd, fs, ft

MIPS64, MIPS32 Release 2

Purpose: Floating Point Subtract

To subtract FP values

Description: FPR[fd] \leftarrow FPR[fs] - FPR[ft]

The value in FPR *ft* is subtracted from the value in FPR *fs*. The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. SUB.PS subtracts the upper and lower halves of FPR *fs* and FPR *ft* independently, and ORs together any generated exceptional conditions.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*. If they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of SUB.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
StoreFPR (fd, fmt, ValueFPR(fs, fmt) -fmt ValueFPR(ft, fmt))
```

CPU Exceptions:

Coprocessor Unusable, Reserved Instruction

FPU Exceptions:

Inexact, Overflow, Underflow, Invalid Op, Unimplemented Op

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	SUBU 100011	

6 5 5 5 5 6 0

Format: SUBU rd, rs, rt**MIPS32****Purpose:** Subtract Unsigned Word

To subtract 32-bit integers

Description: $GPR[rd] \leftarrow GPR[rs] - GPR[rt]$ The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* and the 32-bit arithmetic result is sign-extended and placed into GPR *rd*.

No integer overflow exception occurs under any circumstances.

Restrictions:On 64-bit processors, if either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.**Operation:**

```

if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← GPR[rs] - GPR[rt]
GPR[rd] ← sign_extend(temp)

```

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

31	26 25	21 20	16 15	11 10	6 5	0
COP1X 010011	base	index	fs	0 00000	SUXC1 001101	6

6 5 5 5 5 6

Format: SUXC1 fs, index(base)**MIPS64, MIPS32 Release 2****Purpose:** Store Doubleword Indexed Unaligned from Floating Point

To store a doubleword from an FPR to memory (GPR+GPR addressing) ignoring alignment

Description: `memory[(GPR[base] + GPR[index])PSIZE-1..3] ← FPR[fs]`

The contents of the 64-bit doubleword in FPR *fs* is stored at the memory location specified by the effective address. The contents of GPR *index* and GPR *base* are added to form the effective address. The effective address is double-word-aligned; EffectiveAddress_{2..0} are ignored.

Restrictions:

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
vAddr ← (GPR[base]+GPR[index])63..3 || 03
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Watch

31	26 25	21 20	16 15	0
SW 101011	base	rt	offset	

6 5 5 16

Format: SW rt, offset(base)**MIPS32****Purpose:** Store Word

To store a word to memory

Description: memory[GPR[base] + offset] \leftarrow GPR[rt]The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.**Restrictions:**

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```

vAddr  $\leftarrow$  sign_extend(offset) + GPR[base]
if vAddr1..0  $\neq$  02 then
    SignalException(AddressError)
endif
(pAddr, CCA)  $\leftarrow$  AddressTranslation(vAddr, DATA, STORE)
pAddr  $\leftarrow$  pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
bytesel  $\leftarrow$  vAddr2..0 xor (BigEndianCPU || 02)
datadoubleword  $\leftarrow$  GPR[rt]63-8*bytesel..0 || 08*bytesel
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

31	26 25	21 20	16 15	0
SWC1 111001	base	ft	offset	

SWC1 ft, offset(base)

MIPS32

Purpose: Store Word from Floating Point

To store a word from an FPR to memory

Description: `memory[GPR[base] + offset] ← FPR[ft]`The low 32-bit word from FPR *ft* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.**Restrictions:**An Address Error exception occurs if EffectiveAddress_{1..0} ≠ 0 (not word-aligned).**Operation:**

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_WORD) || 08*bytesel
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

31	26 25	21 20	16 15	0
SWC2 111010	base	rt	offset	

6 5 5 16

Format: SWC2 rt, offset(base)**MIPS32****Purpose:** Store Word from Coprocessor 2

To store a word from a COP2 register to memory

Description: $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{CPR}[2, \text{rt}, 0]$ The low 32-bit word from COP2 (Coprocessor 2) register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.**Restrictions:**An Address Error exception occurs if EffectiveAddress_{1..0} ≠ 0 (not word-aligned).**Operation:**

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
datadoubleword ← CPR[2, rt, 0]63-8*bytesel..0 || 08*bytesel
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

31	26 25	21 20	16 15		7 6 5	0
SPECIAL3 011111	base	rt		offset	0	SWE 011111

6 5 5 9 1 6

Format: SWE rt, offset(base)

MIPS32

Purpose:

To store a word to user mode virtual address space when executing in kernel mode.

Description:

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The SWE instruction functions in exactly the same fashion as the SW instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the *Config5_{EVA}* field being set to one.

Restrictions:

Only usable in kernel mode when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

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]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
datadoubleword ← GPR[rt]63-8*bytesel..0 || 08*bytesel
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill

TLB Invalid

Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable

31	26 25	21 20	16 15	0
SWL 101010	base	rt		offset

6 5 5 16

Format: SWL rt, offset(base)**MIPS32****Purpose:** Store Word Left

To store the most-significant part of a word to an unaligned memory address

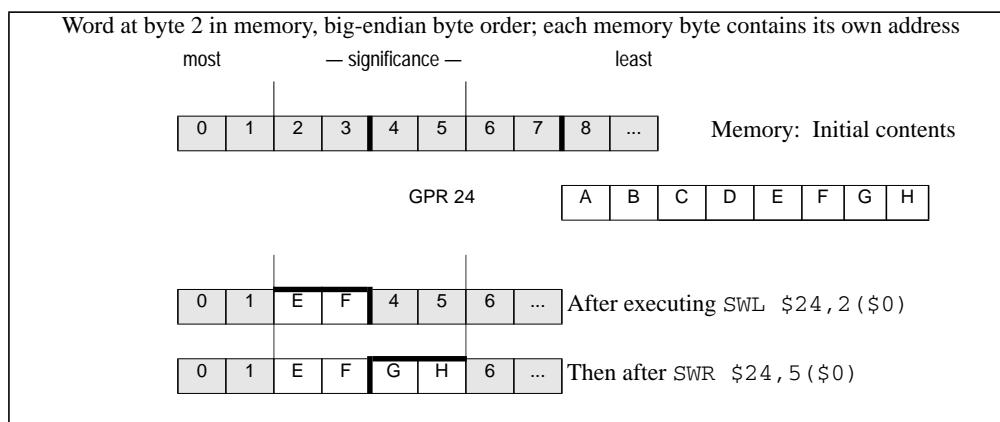
Description: $\text{memory}[\text{GPR}[base] + \text{offset}] \leftarrow \text{GPR}[rt]$

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

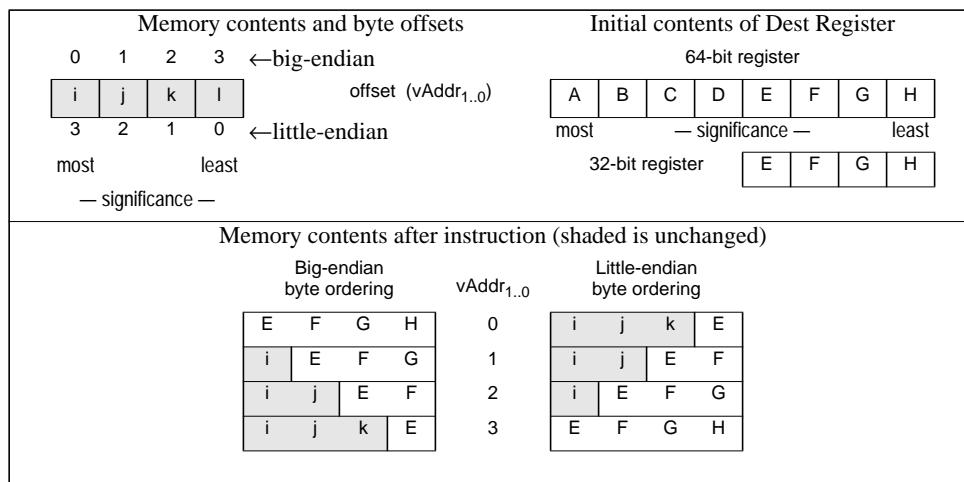
A part of *W*, the most-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. The same number of the most-significant (left) bytes from the word in GPR *rt* are stored into these bytes of *W*.

If GPR *rt* is a 64-bit register, the source word is the low word of the register.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWL stores the most-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWR stores the remainder of the unaligned word.

Figure 3.28 Unaligned Word Store Using SWL and SWR

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address ($vAddr_{1..0}$)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte ordering.

Figure 3.29 Bytes Stored by an SWL Instruction**Restrictions:**

None

Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
If BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..2 || 02
endif
byte ← vAddr1..0 xor BigEndianCPU2
if (vAddr2 xor BigEndianCPU) = 0 then
    datadoubleword ← 032 || 024-8*byte || GPR[rt]31..24-8*byte
else
    datadoubleword ← 024-8*byte || GPR[rt]31..24-8*byte || 032
endif

StoreMemory(CCA, byte, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch

31	26 25	21 20	16 15	7 6 5	0
SPECIAL3 011111	base	rt	offset	0	SWLE 100001

6 5 5 9 1 6

Format: SWLE rt, offset(base)

MIPS32

Purpose: Store Word Left EVA

To store the most-significant part of a word to an unaligned user mode virtual address while operating in kernel mode.

Description: memory[GPR[base] + offset] \leftarrow GPR[rt]

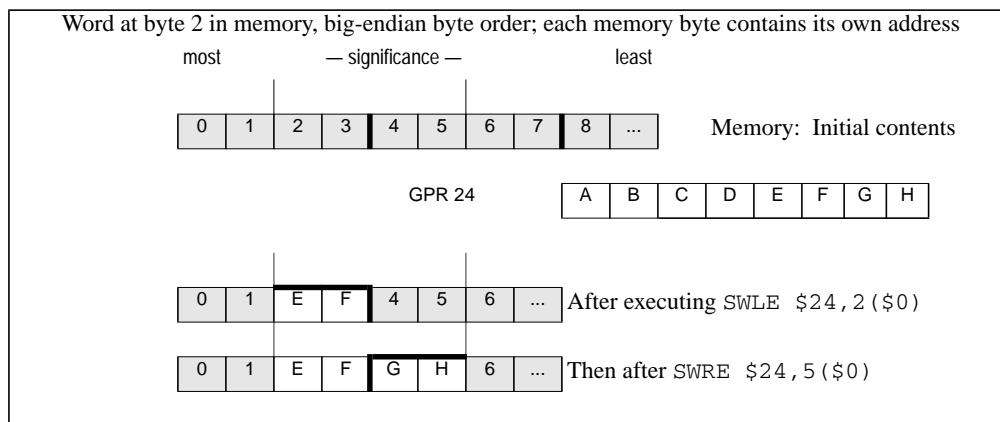
The 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

A part of *W*, the most-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. The same number of the most-significant (left) bytes from the word in GPR *rt* are stored into these bytes of *W*.

If GPR *rt* is a 64-bit register, the source word is the low word of the register.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWLE stores the most-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWRE stores the remainder of the unaligned word.

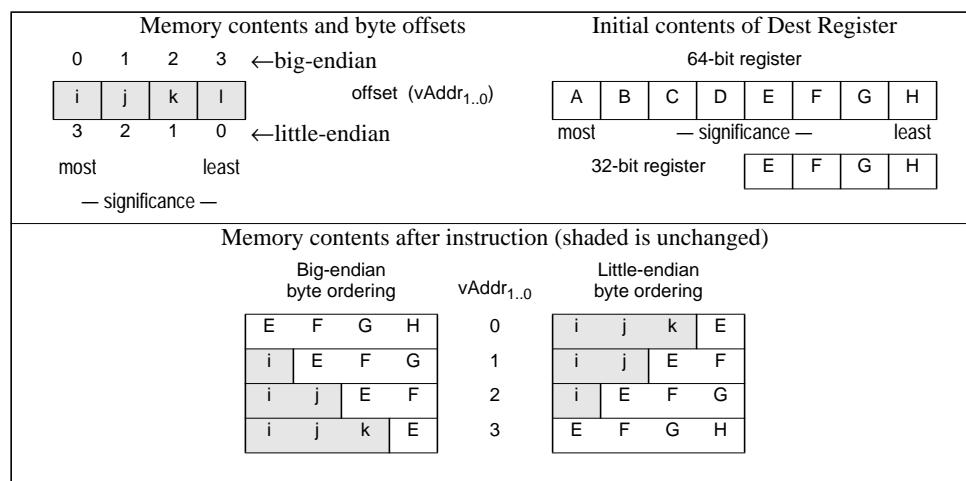
Figure 3.30 Unaligned Word Store Using SWLE and SWRE



The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address ($vAddr_{1..0}$)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte ordering.

The SWLE instruction functions in exactly the same fashion as the SWL instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Figure 3.31 Bytes Stored by an SWLE Instruction**Restrictions:**

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Operation:

```

 $vAddr \leftarrow sign\_extend(offset) + GPR[base]$ 
 $(pAddr, CCA) \leftarrow AddressTranslation(vAddr, DATA, STORE)$ 
 $pAddr \leftarrow pAddr_{PSIZE-1..3} || (pAddr_{2..0} \oplus ReverseEndian^3)$ 
If BigEndianMem = 0 then
     $pAddr \leftarrow pAddr_{PSIZE-1..2} || 0^2$ 
endif
byte  $\leftarrow vAddr_{1..0} \oplus BigEndianCPU^2$ 
if  $(vAddr_2 \oplus BigEndianCPU) = 0$  then
    datadoubleword  $\leftarrow 0^{32} || 0^{24-8*byte} || GPR[rt]_{31..24-8*byte}$ 
else
    datadoubleword  $\leftarrow 0^{24-8*byte} || GPR[rt]_{31..24-8*byte} || 0^{32}$ 
endif
StoreMemory(CCA, byte, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error , Watch, Reserved Instruction, Coprocessor Unusable

31	26 25	21 20	16 15	0
SWR 101110	base	rt	offset	

6 5 5 16

Format: SWR rt, offset(base)

MIPS32

Purpose: Store Word Right

To store the least-significant part of a word to an unaligned memory address

Description: $\text{memory}[\text{GPR}[base] + \text{offset}] \leftarrow \text{GPR}[rt]$

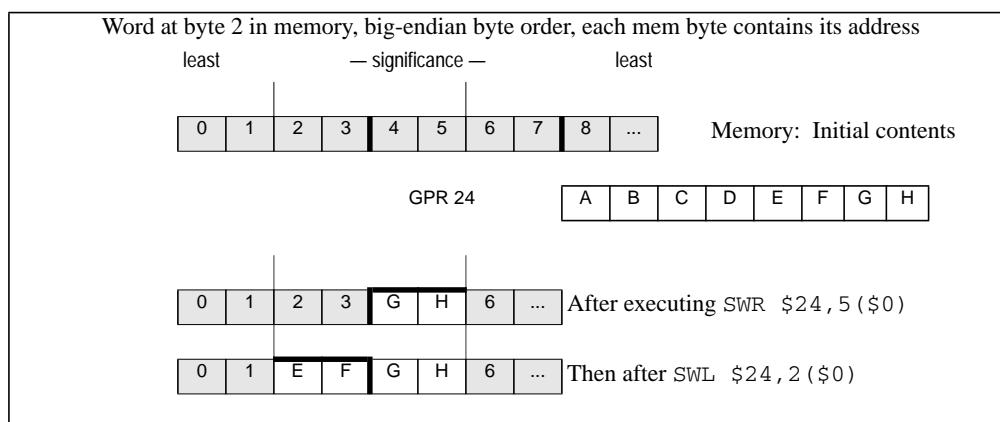
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

A part of *W*, the least-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. The same number of the least-significant (right) bytes from the word in GPR *rt* are stored into these bytes of *W*.

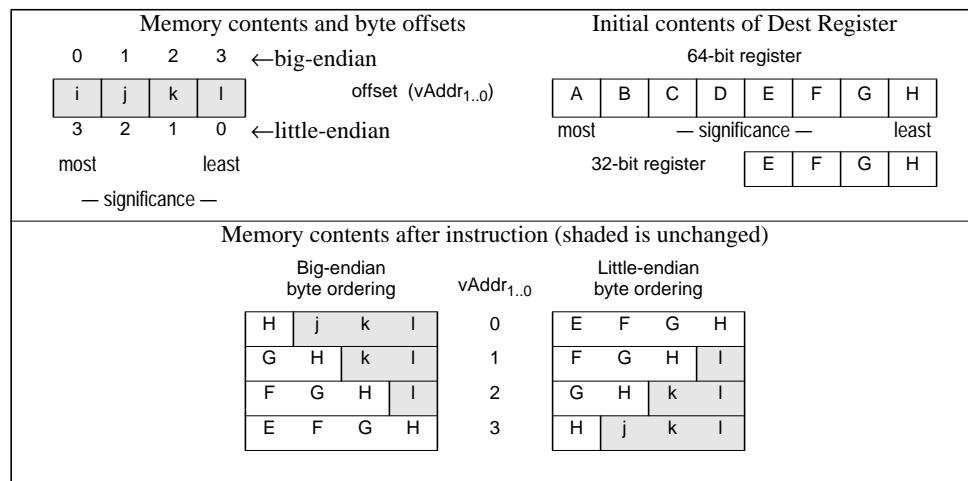
If GPR *rt* is a 64-bit register, the source word is the low word of the register.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR stores the least-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWL stores the remainder of the unaligned word.

Figure 3.32 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 3.33 Bytes Stored by SWR Instruction**Restrictions:**

None

Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
If BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..2 || 02
endif
byte ← vAddr1..0 xor BigEndianCPU2
if (vAddr2 xor BigEndianCPU) = 0 then
    datadoubleword ← 032 || GPR[rt]31-8*byte..0 || 08*byte
else
    datadoubleword ← GPR[rt]31-8*byte..0 || 08*byte || 032
endif

StoreMemory(CCA, WORD-byte, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch

31	26 25	21 20	16 15	7 6 5	0
SPECIAL3 011111	base	rt	offset	0	SWRE 100010

6 5 5 9 1 6

Format: SWRE rt, offset(base)

MIPS32

Purpose: Store Word Right EVA

To store the least-significant part of a word to an unaligned user mode virtual address while operating in kernel mode.

Description: $\text{memory}[\text{GPR}[base] + \text{offset}] \leftarrow \text{GPR}[rt]$

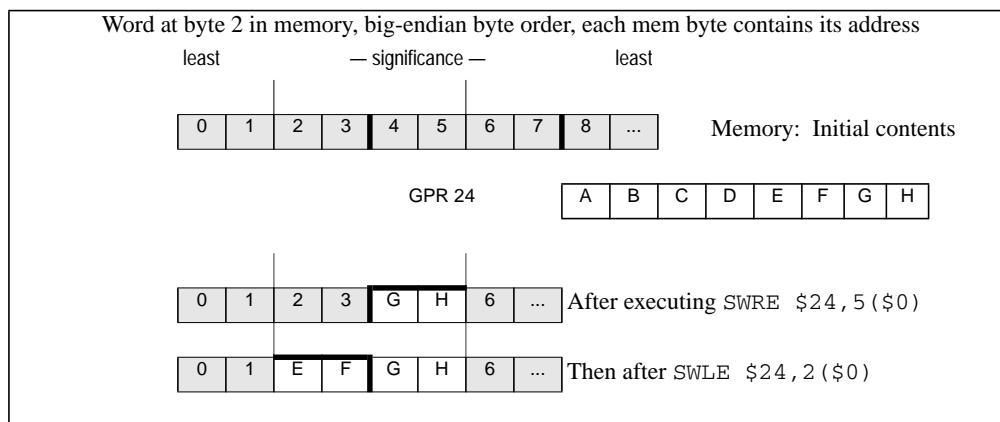
The 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

A part of *W*, the least-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. The same number of the least-significant (right) bytes from the word in GPR *rt* are stored into these bytes of *W*.

If GPR *rt* is a 64-bit register, the source word is the low word of the register.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWRE stores the least-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWLE stores the remainder of the unaligned word.

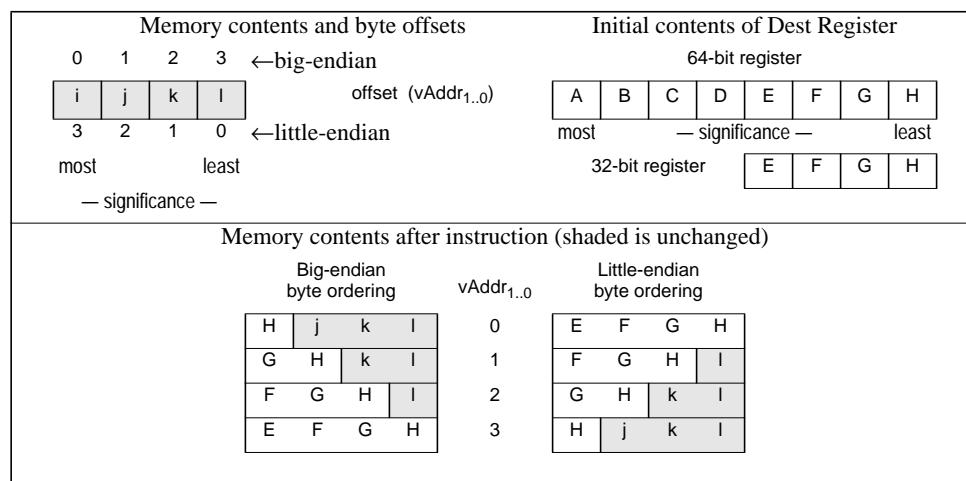
Figure 3.34 Unaligned Word Store Using SWRE and SWLE



The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address ($vAddr_{l..0}$)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte-ordering.

The LWE instruction functions in exactly the same fashion as the LW instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Figure 3.35 Bytes Stored by SWRE Instruction**Restrictions:**

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
If BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..2 || 02
endif
byte ← vAddr1..0 xor BigEndianCPU2
if (vAddr2 xor BigEndianCPU) = 0 then
    datadoubleword ← 032 || GPR[rt]31-8*byte..0 || 08*byte
else
    datadoubleword ← GPR[rt]31-8*byte..0 || 08*byte || 032
endif
StoreMemory(CCA, WORD-byte, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch, Coprocessor Unusable

31	26 25	21 20	16 15	11 10	6 5	0
COP1X 010011	base	index	fs	0 00000	SWXC1 001000	6

6 5 5 5 5 6

Format: SWXC1 fs, index(base)**MIPS64**
MIPS32 Release 2**Purpose:** Store Word Indexed from Floating Point

To store a word from an FPR to memory (GPR+GPR addressing)

Description: memory[GPR[base] + GPR[index]] ← FPR[fs]The low 32-bit word from FPR *fs* is stored in memory at the location specified by the aligned effective address. The contents of GPR *index* and GPR *base* are added to form the effective address.**Restrictions:**An Address Error exception occurs if EffectiveAddress_{1..0} ≠ 0 (not word-aligned).**Operation:**

```

vAddr ← GPR[base] + GPR[index]
if vAddr1..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
datadoubleword ← ValueFPR(fs, UNINTERPRETED_WORD) || 08*bytesel
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Reserved Instruction, Coprocessor Unusable, Watch

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000		0 00 0000 0000 0000 0		stype		SYNC 001111

Format: SYNC (stype = 0 implied)
SYNC stype

MIPS32
MIPS32

Purpose: To order loads and stores for shared memory.

Description:

These types of ordering guarantees are available through the SYNC instruction:

- Completion Barriers
- Ordering Barriers

Simple Description for Completion Barrier:

- The barrier affects only *uncached* and *cached coherent* loads and stores.
- The specified memory instructions (loads or stores or both) that occur before the SYNC instruction must be completed before the specified memory instructions after the SYNC are allowed to start.
- Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.

Detailed Description for Completion Barrier:

- Every synchronizable specified memory instruction (loads or stores or both) that occurs in the instruction stream before the SYNC instruction must be already globally performed before any synchronizable specified memory instructions that occur after the SYNC are allowed to be performed, with respect to any other processor or coherent I/O module.
- The barrier does not guarantee the order in which instruction fetches are performed.
- A stype value of zero will always be defined such that it performs the most complete set of synchronization operations that are defined. This means stype zero always does a completion barrier that affects both loads and stores preceding the SYNC instruction and both loads and stores that are subsequent to the SYNC instruction. Non-zero values of stype may be defined by the architecture or specific implementations to perform synchronization behaviors that are less complete than that of stype zero. If an implementation does not use one of these non-zero values to define a different synchronization behavior, then that non-zero value of stype must act the same as stype zero completion barrier. This allows software written for an implementation with a lighter-weight barrier to work on another implementation which only implements the stype zero completion barrier.
- A completion barrier is required, potentially in conjunction with SSNOP (in Release 1 of the Architecture) or EHB (in Release 2 of the Architecture), to guarantee that memory reference results are visible across operating mode changes. For example, a completion barrier is required on some implementations on entry to and exit from Debug Mode to guarantee that memory effects are handled correctly.

SYNC behavior when the stype field is zero:

- A completion barrier that affects preceding loads and stores and subsequent loads and stores.

Simple Description for Ordering Barrier:

- The barrier affects only *uncached* and *cached coherent* loads and stores.
- The specified memory instructions (loads or stores or both) that occur before the SYNC instruction must always be ordered before the specified memory instructions after the SYNC.
- Memory instructions which are ordered before other memory instructions are processed by the load/store datapath first before the other memory instructions.

Detailed Description for Ordering Barrier:

- Every synchronizable specified memory instruction (loads or stores or both) that occurs in the instruction stream before the SYNC instruction must reach a stage in the load/store datapath after which no instruction re-ordering is possible before any synchronizable specified memory instruction which occurs after the SYNC instruction in the instruction stream reaches the same stage in the load/store datapath.
- If any memory instruction before the SYNC instruction in program order, generates a memory request to the external memory and any memory instruction after the SYNC instruction in program order also generates a memory request to external memory, the memory request belonging to the older instruction must be globally performed before the time the memory request belonging to the younger instruction is globally performed.
- The barrier does not guarantee the order in which instruction fetches are performed.

As compared to the completion barrier, the ordering barrier is a lighter-weight operation as it does not require the specified instructions before the SYNC to be already completed. Instead it only requires that those specified instructions which are subsequent to the SYNC in the instruction stream are never re-ordered for processing ahead of the specified instructions which are before the SYNC in the instruction stream. This potentially reduces how many cycles the barrier instruction must stall before it completes.

The Acquire and Release barrier types are used to minimize the memory orderings that must be maintained and still have software synchronization work.

Implementations that do not use any of the non-zero values of stype to define different barriers, such as ordering barriers, must make those stype values act the same as stype zero.

For the purposes of this description, the CACHE, PREF and PREFIX instructions are treated as loads and stores. That is, these instructions and the memory transactions sourced by these instructions obey the ordering and completion rules of the SYNC instruction.

Table 3.36 lists the available completion barrier and ordering barriers behaviors that can be specified using the stype field..

Table 3.36 Encodings of the Bits[10:6] of the SYNC instruction; the SType Field

Code	Name	Older instructions which must reach the load/store ordering point before the SYNC instruction completes.	Younger instructions which must reach the load/store ordering point only after the SYNC instruction completes.	Older instructions which must be globally performed when the SYNC instruction completes	Compliance
0x0	SYNC or SYNC 0	Loads, Stores	Loads, Stores	Loads, Stores	Required
0x4	SYNC_WMB or SYNC 4	Stores	Stores		Optional
0x10	SYNC_MB or SYNC 16	Loads, Stores	Loads, Stores		Optional
0x11	SYNC_ACQUIRE or SYNC 17	Loads	Loads, Stores		Optional
0x12	SYNC_RELEASE or SYNC 18	Loads, Stores	Stores		Optional
0x13	SYNC_RMB or SYNC 19	Loads	Loads		Optional
0x1-0x3, 0x5-0xF					Implementation-Specific and Vendor Specific Sync Types
0x14 - 0x1F	RESERVED				Reserved for MIPS Technologies for future extension of the architecture.

Terms:

Synchronizable: A load or store instruction is *synchronizable* if the load or store occurs to a physical location in shared memory using a virtual location with a memory access type of either *uncached* or *cached coherent*. *Shared memory* is memory that can be accessed by more than one processor or by a coherent I/O system module.

Performed load: A load instruction is *performed* when the value returned by the load has been determined. The result of a load on processor A has been *determined* with respect to processor or coherent I/O module B when a subsequent

store to the location by B cannot affect the value returned by the load. The store by B must use the same memory access type as the load.

Performed store: A store instruction is *performed* when the store is observable. A store on processor A is *observable* with respect to processor or coherent I/O module B when a subsequent load of the location by B returns the value written by the store. The load by B must use the same memory access type as the store.

Globally performed load: A load instruction is *globally performed* when it is performed with respect to all processors and coherent I/O modules capable of storing to the location.

Globally performed store: A store instruction is *globally performed* when it is globally observable. It is *globally observable* when it is observable by all processors and I/O modules capable of loading from the location.

Coherent I/O module: A *coherent I/O module* is an Input/Output system component that performs coherent Direct Memory Access (DMA). It reads and writes memory independently as though it were a processor doing loads and stores to locations with a memory access type of *cached coherent*.

Load/Store Datapath: The portion of the processor which handles the load/store data requests coming from the processor pipeline and processes those requests within the cache and memory system hierarchy.

Restrictions:

The effect of SYNC on the global order of loads and stores for memory access types other than *uncached* and *cached coherent* is **UNPREDICTABLE**.

Operation:

```
SyncOperation(stype)
```

Exceptions:

None

Programming Notes:

A processor executing load and store instructions observes the order in which loads and stores using the same memory access type occur in the instruction stream; this is known as *program order*.

A *parallel program* has multiple instruction streams that can execute simultaneously on different processors. In multiprocessor (MP) systems, the order in which the effects of loads and stores are observed by other processors—the *global order* of the loads and store—determines the actions necessary to reliably share data in parallel programs.

When all processors observe the effects of loads and stores in program order, the system is *strongly ordered*. On such systems, parallel programs can reliably share data without explicit actions in the programs. For such a system, SYNC has the same effect as a NOP. Executing SYNC on such a system is not necessary, but neither is it an error.

If a multiprocessor system is not strongly ordered, the effects of load and store instructions executed by one processor may be observed out of program order by other processors. On such systems, parallel programs must take explicit actions to reliably share data. At critical points in the program, the effects of loads and stores from an instruction stream must occur in the same order for all processors. SYNC separates the loads and stores executed on the processor into two groups, and the effect of all loads and stores in one group is seen by all processors before the effect of any load or store in the subsequent group. In effect, SYNC causes the system to be strongly ordered for the executing processor at the instant that the SYNC is executed.

Many MIPS-based multiprocessor systems are strongly ordered or have a mode in which they operate as strongly ordered for at least one memory access type. The MIPS architecture also permits implementation of MP systems that are not strongly ordered; SYNC enables the reliable use of shared memory on such systems. A parallel program that does not use SYNC generally does not operate on a system that is not strongly ordered. However, a program that does use SYNC works on both types of systems. (System-specific documentation describes the actions needed to reliably

share data in parallel programs for that system.)

The behavior of a load or store using one memory access type is **UNPREDICTABLE** if a load or store was previously made to the same physical location using a different memory access type. The presence of a SYNC between the references does not alter this behavior.

SYNC affects the order in which the effects of load and store instructions appear to all processors; it does not generally affect the physical memory-system ordering or synchronization issues that arise in system programming. The effect of SYNC on implementation-specific aspects of the cached memory system, such as writeback buffers, is not defined.

```
# Processor A (writer)
# Conditions at entry:
# The value 0 has been stored in FLAG and that value is observable by B
SW      R1, DATA          # change shared DATA value
LI      R2, 1
SYNC
SW      R2, FLAG          # say that the shared DATA value is valid

# Processor B (reader)
LI      R2, 1
1: LW      R1, FLAG    # Get FLAG
BNE    R2, R1, 1B# if it says that DATA is not valid, poll again
NOP
SYNC          # FLAG value checked before doing DATA read
LW      R1, DATA    # Read (valid) shared DATA value
```

The code fragments above shows how SYNC can be used to coordinate the use of shared data between separate writer and reader instruction streams in a multiprocessor environment. The FLAG location is used by the instruction streams to determine whether the shared data item DATA is valid. The SYNC executed by processor A forces the store of DATA to be performed globally before the store to FLAG is performed. The SYNC executed by processor B ensures that DATA is not read until after the FLAG value indicates that the shared data is valid.

Software written to use a SYNC instruction with a non-zero stype value, expecting one type of barrier behavior, should only be run on hardware that actually implements the expected barrier behavior for that non-zero stype value or on hardware which implements a superset of the behavior expected by the software for that stype value. If the hardware does not perform the barrier behavior expected by the software, the system may fail.

31	26 25	21 20	16 15	0
REGIMM 000001	base	SYNCI 11111	offset	

6 5 5 16

Format: SYNCI offset (base)

MIPS32 Release 2

Purpose: Synchronize Caches to Make Instruction Writes Effective

To synchronize all caches to make instruction writes effective.

Description:

This instruction is used after a new instruction stream is written to make the new instructions effective relative to an instruction fetch, when used in conjunction with the SYNC and JALR.HB, JR.HB, or ERET instructions, as described below. Unlike the CACHE instruction, the SYNCI instruction is available in all operating modes in an implementation of Release 2 of the architecture.

The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used to address the cache line in all caches which may need to be synchronized with the write of the new instructions. The operation occurs only on the cache line which may contain the effective address. One SYNCI instruction is required for every cache line that was written. See the Programming Notes below.

A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur as a byproduct of this instruction. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS. This instruction never causes Execute-Inhibit nor Read-Inhibit exceptions.

A Cache Error exception may occur as a byproduct of this instruction. For example, if a writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error.

An Address Error Exception (with cause code equal AdEL) may occur if the effective address references a portion of the kernel address space which would normally result in such an exception. It is implementation dependent whether such an exception does occur.

It is implementation dependent whether a data watch is triggered by a SYNCI instruction whose address matches the Watch register address match conditions. In multiprocessor implementations where instruction caches are not coherently maintained by hardware, the SYNCI instruction may optionally affect all coherent icaches within the system. If the effective address uses a coherent Cacheability and Coherency Attribute (CCA), then the operation may be *globalized*, meaning it is broadcast to all of the coherent instruction caches within the system. If the effective address does not use one of the coherent CCAs, there is no broadcast of the SYNCI operation. If multiple levels of caches are to be affected by one SYNCI instruction, all of the affected cache levels must be processed in the same manner - either all affected cache levels use the globalized behavior or all affected cache levels use the non-globalized behavior.

In multiprocessor implementations where instruction caches are coherently maintained by hardware, the SYNCI instruction should behave as a NOP instruction.

Restrictions:

The operation of the processor is **UNPREDICTABLE** if the effective address references any instruction cache line that contains instructions to be executed between the SYNCI and the subsequent JALR.HB, JR.HB, or ERET instruction required to clear the instruction hazard.

The SYNCI instruction has no effect on cache lines that were previously locked with the CACHE instruction. If correct software operation depends on the state of a locked line, the CACHE instruction must be used to synchronize the caches.

The SYNCI instruction acts on the current processor at a minimum. It is implementation specific whether it affects

the caches on other processors in a multi-processor system, except as required to perform the operation on the current processor (as might be the case if multiple processors share an L2 or L3 cache).

Full visibility of the new instruction stream requires execution of a subsequent SYNC instruction, followed by a JALR.HB, JR.HB, DERET, or ERET instruction. The operation of the processor is **UNPREDICTABLE** if this sequence is not followed.

Operation:

```
vaddr ← GPR[base] + sign_extend(offset)
SynchronizeCacheLines(vaddr)      /* Operate on all caches */
```

Exceptions:

Reserved Instruction Exception (Release 1 implementations only)

TLB Refill Exception

TLB Invalid Exception

Address Error Exception

Cache Error Exception

Bus Error Exception

Programming Notes:

When the instruction stream is written, the SYNCI instruction should be used in conjunction with other instructions to make the newly-written instructions effective. The following example shows a routine which can be called after the new instruction stream is written to make those changes effective. Note that the SYNCI instruction could be replaced with the corresponding sequence of CACHE instructions (when access to Coprocessor 0 is available), and that the JR.HB instruction could be replaced with JALR.HB, ERET, or DERET instructions, as appropriate. A SYNC instruction is required between the final SYNCI instruction in the loop and the instruction that clears instruction hazards.

```
/*
 * This routine makes changes to the instruction stream effective to the
 * hardware. It should be called after the instruction stream is written.
 * On return, the new instructions are effective.
 *
 * Inputs:
 *      a0 = Start address of new instruction stream
 *      a1 = Size, in bytes, of new instruction stream
 */

    beq    a1, zero, 20f      /* If size==0, */
    nop                /* branch around */
    addu   a1, a0, a1      /* Calculate end address + 1 */
                           /* (daddu for 64-bit addressing) */
    rdhwr  v0, HW_SYNCI_Step /* Get step size for SYNCI from new */
                           /* Release 2 instruction */
    beq    v0, zero, 20f      /* If no caches require synchronization, */
    nop                /* branch around */
10: synci  0(a0)          /* Synchronize all caches around address */
    addu   a0, a0, v0      /* Add step size in delay slot */
                           /* (daddu for 64-bit addressing) */
    sltu   v1, a0, a1      /* Compare current with end address */
    bne    v1, zero, 10b    /* Branch if more to do */
    nop                /* branch around */
    sync               /* Clear memory hazards */
20: jr.hb  ra            /* Return, clearing instruction hazards */
    nop
```

31	26 25		6 5	0
SPECIAL 000000		code	SYSCALL 001100	6

6 20 6

Format: SYSCALL**MIPS32****Purpose:** System Call

To cause a System Call exception

Description:

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Restrictions:

None

Operation:

```
SignalException(SystemCall)
```

Exceptions:

System Call

31	26 25	21 20	16 15	6 5	0
SPECIAL 000000	rs	rt	code	TEQ 110100	6

6 5 5 10 6

Format: TEQ rs, rt**MIPS32****Purpose:** Trap if Equal

To compare GPRs and do a conditional trap

Description: if GPR[rs] = GPR[rt] then TrapCompare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is equal to GPR *rt*, then take a Trap exception.The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.**Restrictions:**

None

Operation:

```

if GPR[rs] = GPR[rt] then
    SignalException(Trap)
endif

```

Exceptions:

Trap

31	26 25	21 20	16 15	0
REGIMM 000001	rs	TEQI 01100		immediate

6 5 5 16

Format: TEQI rs, immediate**MIPS32****Purpose:** Trap if Equal Immediate

To compare a GPR to a constant and do a conditional trap

Description: if GPR[rs] = immediate then TrapCompare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is equal to *immediate*, then take a Trap exception.**Restrictions:**

None

Operation:

```

if GPR[rs] = sign_extend(immediate) then
    SignalException(Trap)
endif

```

Exceptions:

Trap

31	26 25	21 20	16 15	6 5	0
SPECIAL 000000	rs	rt	code	TGE 110000	6

Format: TGE rs, rt**MIPS32****Purpose:** Trap if Greater or Equal

To compare GPRs and do a conditional trap

Description: if $\text{GPR}[rs] \geq \text{GPR}[rt]$ then TrapCompare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is greater than or equal to GPR *rt*, then take a Trap exception.The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.**Restrictions:**

None

Operation:

```

if GPR[rs] ≥ GPR[rt] then
    SignalException(Trap)
endif

```

Exceptions:

Trap

31	26 25	21 20	16 15	0
REGIMM 000001	rs	TGEI 01000		immediate

6 5 5 16

Format: TGEI rs, immediate**MIPS32****Purpose:** Trap if Greater or Equal Immediate

To compare a GPR to a constant and do a conditional trap

Description: if GPR[rs] \geq immediate then TrapCompare 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]  $\geq$  sign_extend(immediate) then
    SignalException(Trap)
endif

```

Exceptions:

Trap

31	26 25	21 20	16 15	0
REGIMM 000001	rs	TGEIU 01001		immediate

6 5 5 16

Format: TGEIU rs, immediate**MIPS32****Purpose:** Trap if Greater or Equal Immediate Unsigned

To compare a GPR to a constant and do a conditional trap

Description: if GPR[rs] \geq immediate then TrapCompare 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])  $\geq$  (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif

```

Exceptions:

Trap

31	26 25	21 20	16 15	6 5	0
SPECIAL 000000	rs	rt	code	TGEU 110001	6

Format: TGEU rs, rt**MIPS32****Purpose:** Trap if Greater or Equal Unsigned

To compare GPRs and do a conditional trap

Description: if $GPR[rs] \geq GPR[rt]$ then TrapCompare 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

31	26	25	24		6	5	0
COP0 010000	CO 1			0 000 0000 0000 0000 0000		TLBINV 000011	6

6 1 19 6

Format: TLBINV**MIPS32****Purpose:** TLB Invalidate

TLBINV invalidates a set of TLB entries based on ASID and *Index* match. The virtual address is ignored in the entry match. TLB entries which have their G bit set to 1 are not modified.

Implementation of the TLBINV instruction is optional. The implementation of this instruction is indicated by the IE field in *Config4*.

Implementation of *EntryHi_{EHINV}* field is required for implementation of TLBGINV instruction.

Support for TLBINV is recommended for implementations supporting VTLB/FTLB type of MMU.

Description:

On execution of the TLBINV instruction, the set of TLB entries with matching ASID are marked invalid, excluding those TLB entries which have their G bit set to 1.

The *EntryHi_{ASID}* field has to be set to the appropriate ASID value before executing the TLBINV instruction.

Behavior of the TLBINV instruction applies to all applicable TLB entries and is unaffected by the setting of the *Wired* register.

For JTLB-based MMU(*Config_{MT}*=1):

All matching entries in the JTLB are invalidated. *Index* is unused.

For VTLB/FTLB -based MMU(*Config_{MT}*=4):

A TLBINV with *Index* set in VTLB range causes all matching entries in the VTLB to be invalidated.

A TLBINV with *Index* set in FTLB range causes all matching entries in the single corresponding FTLB set to be invalidated.

If TLB invalidate walk is implemented in software (*Config4_{IE}*=2), then software must do these steps to flush the entire MMU:

1. one TLBINV instruction is executed with an index in VTLB range (invalidates all matching VTLB entries)
2. a TLBINV instruction is executed for each FTLB set (invalidates all matching entries in FTLB set)

If TLB invalidate walk is implemented in hardware (*Config4_{IE}*=3), then software must do these steps to flush the entire MMU:

1. one TLBINV instruction is executed (invalidates all matching entries in both FTLB & VTLB). In this case, *Index* is unused.

Restrictions:

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of available TLB entries (For the case of $Config_{MT}=4$).

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```

if ( ConfigMT=1 or (ConfigMT=4 & Config4IE=2 & Index ≤ Config1MMU_SIZE-1) )
    startnum ← 0
    endnum ← Config1MMU_SIZE-1
endif
// treating VTLB and FTLB as one array
if (ConfigMT=4 & Config4IE=2 & Index > Config1MMU_SIZE-1)
    startnum ← start of selected FTLB set // implementation specific
    endnum ← end of selected FTLB set - 1 //implementation specifc
endif

if (ConfigMT=4 & Config4IE=3))
    startnum ← 0
    endnum ← Config1MMU_SIZE-1 + ((Config4FTLBWays + 2) * Config4FTLBSets)
endif

for (i = startnum to endnum)
    if (TLB[i].ASID = EntryHiASID & TLB[i].G = 0)
        TLB[i]VPN2_invalid ← 1
    endif
endfor

```

Exceptions:

Coprocessor Unusable

31	26	25	24		6	5	0
COP0 010000	CO 1			0 000 0000 0000 0000 0000		TLBINVF 000100	6

6 1 19 6

Format: TLBINVF**MIPS32****Purpose:** TLB Invalidate Flush

TLBINVF invalidates a set of TLB entries based on *Index* match. The virtual address and ASID are ignored in the entry match.

Implementation of the TLBINVF instruction is optional. The implementation of this instruction is indicated by the IE field in *Config4*.

Implementation of the *EntryHi.EHINV* field is required for implementation of TLBINV and TLBINVF instructions.

Support for TLBINVF is recommended for implementations supporting VTLB/FTLB type of MMU.

Description:

On execution of the TLBINVF instruction, all entries within range of *Index* are invalidated.

Behavior of the TLBINVF instruction applies to all applicable TLB entries and is unaffected by the setting of the *Wired* register.

For JTLB-based MMU(*Config_{MT}*=1):

TLBINVF causes all entries in the JTLB to be invalidated. *Index* is unused.

For VTLB/FTLB-based MMU(*Config_{MT}*=4):

TLBINVF with *Index* in VTLB range causes all entries in the VTLB to be invalidated.

TLBINVF with *Index* in FTLB range causes all entries in the single corresponding set in the FTLB to be invalidated.

If TLB invalidate walk is implemented in software (*Config4_{IE}*=2), then software must do these steps to flush the entire MMU:

1. one TLBINV instruction is executed with an index in VTLB range (invalidates all VTLB entries)
2. a TLBINV instruction is executed for each FTLB set (invalidates all entries in FTLB set)

If TLB invalidate walk is implemented in hardware (*Config4_{IE}*=3), then software must do these steps to flush the entire MMU:

1. one TLBINV instruction is executed (invalidates all entries in both FTLB & VTLB). In this case, *Index* is unused.

Restrictions:

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of available TLB entries (*Config4_{IE}*=2).

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
if ( ConfigMT=1 or (ConfigMT=4 & Config4IE=2 & Index ≤ Config1MMU_SIZE-1) )
    startnum ← 0
    endnum ← Config1MMU_SIZE-1
endif
// treating VTLB and FTLB as one array
if (ConfigMT=4 & Config4IE=2 & Index > Config1MMU_SIZE-1)
    startnum ← start of selected FTLB set // implementation specific
    endnum ← end of selected FTLB set - 1 //implementation specific
endif

if (ConfigMT=4 & Config4IE=3)
    startnum ← 0
    endnum ← Config1MMU_SIZE-1 + ((Config4FTLBWays + 2) * Config4FTLBSets)
endif

for (i = startnum to endnum)
    TLB[i]VPN2_invalid ← 1
endfor
```

Exceptions:

Coprocessor Unusable

31	26	25	24		6	5	0
COP0 010000	CO 1			0 000 0000 0000 0000 0000		TLBP 001000	

6 1 19 6

Format: TLBP**MIPS32****Purpose:** Probe TLB for Matching Entry

To find a matching entry in the TLB.

Description:

The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the *Index* register is set. In Release 1 of the Architecture, it is implementation dependent whether multiple TLB matches are detected on a TLBP. However, implementations are strongly encouraged to report multiple TLB matches only on a TLB write. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. In Release 3 of the Architecture, multiple TLB matches may be reported on either TLB write or TLB probe.

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```

Index ← 1 || UNPREDICTABLE31
for i in 0...TLBEntries-1
    if ((TLB[i]VPN2 and not (TLB[i]Mask) = 
        (EntryHiVPN2 and not (TLB[i]Mask))) and
        (TLB[i]R = EntryHiR) and
        ((TLB[i]G = 1) or (TLB[i]ASID = EntryHiASID)) then
            Index ← i
    endif
endfor

```

Exceptions:

Coprocessor Unusable

Machine Check

31	26	25	24		6	5	0
COP0 010000	CO 1			0 000 0000 0000 0000 0000		TLBR 000001	6

6 1 19 6

Format: TLBR**MIPS32****Purpose:** Read Indexed TLB Entry

To read an entry from the TLB.

Description:

The *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers are loaded with the contents of the TLB entry pointed to by the *Index* register. In Release 1 of the Architecture, it is implementation dependent whether multiple TLB matches are detected on a TLBR. However, implementations are strongly encouraged to report multiple TLB matches only on a TLB write. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. In Release 3 of the Architecture, multiple TLB matches may be detected on a TLBR.

In an implementation supporting TLB entry invalidation ($Config4_{IE} = 2$ or $Config4_{IE} = 3$), reading an invalidated TLB entry causes 0 to be written to *EntryHi*, *EntryLo0*, *EntryLo1* registers and the *PageMask_{MASK}* register field.

Note that the value written to the *EntryHi*, *EntryLo0*, and *EntryLo1* registers may be different from that originally written to the TLB via these registers in that:

- The value returned in the VPN2 field of the *EntryHi* register may 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
if ( (Config4IE = 2 or Config4IE = 3) and TLB[i].VPN2_invalid = 1) then
    PagemaskMASK ← 0
    EntryHi ← 0
    EntryLo1 ← 0
    EntryLo0 ← 0
```

```
EntryHiEHINV ← 1
else
    PageMaskMask ← TLB[i]Mask
    EntryHi ← TLB[i]R || 0Fill ||
        (TLB[i]VPN2 and not TLB[i]Mask) || # Masking implem dependent
        05 || TLB[i]ASID
    EntryLo1 ← 0Fill ||
        (TLB[i]PFN1 and not TLB[i]Mask) || # Masking mplem dependent
        TLB[i]C1 || TLB[i]D1 || TLB[i]V1 || TLB[i]G
    EntryLo0 ← 0Fill ||
        (TLB[i]PFN0 and not TLB[i]Mask) || # Masking mplem dependent
        TLB[i]C0 || TLB[i]D0 || TLB[i]V0 || TLB[i]G
endif
```

Exceptions:

Coprocessor Unusable

Machine Check

31	26	25	24		6	5	0
COP0 010000	CO 1			0 000 0000 0000 0000 0000		TLBWI 000010	6

6 1 19 6

Format: TLBWI**MIPS32****Purpose:** Write Indexed TLB EntryTo write or invalidate a TLB entry indexed by the *Index* register.**Description:**If $Config4_{IE} < 2$ or $EntryHi_{EHINV}=0$:

The TLB entry pointed to by the Index register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers. It is implementation dependent whether multiple TLB matches are detected on a TLBWI. In such an instance, a Machine Check Exception is signaled. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the *PageMask* register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of *PageMask* register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.

If $Config4_{IE} > 1$ and $EntryHi_{EHINV}=1$:

The TLB entry pointed to by the Index register has its VPN2 field marked as invalid. This causes the entry to be ignored on TLB matches for memory accesses. No Machine Check is generated.

Restrictions:

The operation is **UNDEFINED** if the contents of the Index register are greater than or equal to the number of TLB entries in the processor.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
i ← Index
if (Config4IE = 2 or Config4IE = 3) then
    TLB[i].VPN2_invalid ← 0
    if (EntryHiEHINV=1) then
        TLB[i].VPN2_invalid ← 1
        break
    endif
endif
```

```
TLB[i].Mask ← PageMaskMask
TLB[i]R ← EntryHiR
TLB[i]VPN2 ← EntryHiVPN2 and not PageMaskMask # Implementation dependent
TLB[i]ASID ← EntryHiASID
TLB[i]G ← EntryLo1G and EntryLo0G
TLB[i]PFN1 ← EntryLo1PFN and not PageMaskMask # Implementation dependent
TLB[i]C1 ← EntryLo1C
TLB[i]D1 ← EntryLo1D
TLB[i]V1 ← EntryLo1V
TLB[i]PFN0 ← EntryLo0PFN and not PageMaskMask # Implementation dependent
TLB[i]C0 ← EntryLo0C
TLB[i]D0 ← EntryLo0D
TLB[i]V0 ← EntryLo0V
```

Exceptions:

Coprocessor Unusable

Machine Check

31	26	25	24		6	5	0
COP0 010000	CO 1			0 000 0000 0000 0000 0000		TLBWR 000110	6

6 1 19 6

Format: TLBWR**MIPS32****Purpose:** Write Random TLB EntryTo write a TLB entry indexed by the *Random* register.**Description:**

The TLB entry pointed to by the *Random* register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers. It is implementation dependent whether multiple TLB matches are detected on a TLBWR. In such an instance, a Machine Check Exception is signaled. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the *PageMask* register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of *PageMask* register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```

i ← Random
if (Config4IE = 2 or Config4IE = 3) then
    TLB[i]VPN2_invalid ← 0
endif
TLB[i]Mask ← PageMaskMask
TLB[i]R ← EntryHiR
TLB[i]VPN2 ← EntryHiVPN2 and not PageMaskMask # Implementation dependent
TLB[i]ASID ← EntryHiASID
TLB[i]G ← EntryLo1G and EntryLo0G
TLB[i]PFN1 ← EntryLo1PFN and not PageMaskMask # Implementation dependent
TLB[i]C1 ← EntryLo1C
TLB[i]D1 ← EntryLo1D
TLB[i]V1 ← EntryLo1V
TLB[i]PFN0 ← EntryLo0PFN and not PageMaskMask # Implementation dependent
TLB[i]C0 ← EntryLo0C
TLB[i]D0 ← EntryLo0D
TLB[i]V0 ← EntryLo0V

```

Exceptions:

Coprocessor Unusable

Machine Check

31	26 25	21 20	16 15	6 5	0
SPECIAL 000000	rs	rt	code	TLT 110010	6

Format: TLT rs, rt**MIPS32****Purpose:** Trap if Less Than

To compare GPRs and do a conditional trap

Description: if GPR[rs] < GPR[rt] then TrapCompare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is less than GPR *rt*, then take a Trap exception.The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.**Restrictions:**

None

Operation:

```

if GPR[rs] < GPR[rt] then
    SignalException(Trap)
endif

```

Exceptions:

Trap

31	26 25	21 20	16 15	0
REGIMM 000001	rs	TLTI 01010		immediate

6 5 5 16

Format: TLTI rs, immediate**MIPS32****Purpose:** Trap if Less Than Immediate

To compare a GPR to a constant and do a conditional trap

Description: if GPR[rs] < immediate then TrapCompare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is less than *immediate*, then take a Trap exception.**Restrictions:**

None

Operation:

```

if GPR[rs] < sign_extend(immediate) then
    SignalException(Trap)
endif

```

Exceptions:

Trap

31	26 25	21 20	16 15	0
REGIMM 000001	rs	TLTIU 01011	immediate	

6 5 5 16

Format: TLTIU rs, immediate**MIPS32****Purpose:** Trap if Less Than Immediate Unsigned

To compare a GPR to a constant and do a conditional trap

Description: if GPR[rs] < immediate then TrapCompare 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

31	26 25	21 20	16 15	6 5	0
SPECIAL 000000	rs	rt	code	TLTU 110011	6

Format: TLTU rs, rt**MIPS32****Purpose:** Trap if Less Than Unsigned

To compare GPRs and do a conditional trap

Description: if GPR[rs] < GPR[rt] then 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

31	26 25	21 20	16 15	6 5	0
SPECIAL 000000	rs	rt	code	TNE 110110	6

Format: TNE rs, rt**MIPS32****Purpose:** Trap if Not Equal

To compare GPRs and do a conditional trap

Description: if GPR[rs] ≠ GPR[rt] then TrapCompare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is not equal to GPR *rt*, then take a Trap exception.The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.**Restrictions:**

None

Operation:

```

if GPR[rs] ≠ GPR[rt] then
    SignalException(Trap)
endif

```

Exceptions:

Trap

31	26 25	21 20	16 15	0
REGIMM 000001	rs	TNEI 01110	immediate	

6 5 5 16

Format: TNEI rs, immediate**MIPS32****Purpose:** Trap if Not Equal Immediate

To compare a GPR to a constant and do a conditional trap

Description: if GPR[rs] ≠ immediate then 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] ≠ sign_extend(immediate) then
    SignalException(Trap)
endif

```

Exceptions:

Trap

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	0 00000	fs	fd	TRUNC.L 001001	

6 5 5 5 5 6

Format: TRUNC.L fmt
 TRUNC.L S fd, fs
 TRUNC.L D fd, fs

MIPS64, MIPS32 Release 2
 MIPS64, MIPS32 Release 2

Purpose: Floating Point Truncate to Long Fixed Point

To convert an FP value to 64-bit fixed point, rounding toward zero

Description: FPR[fd] \leftarrow convert_and_round(FPR[fs])

The value in FPR *fs*, in format *fmt*, is converted to a value in 64-bit long fixed point format and rounded toward zero (rounding mode 1). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}-1$, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact

31	26 25	21 20	16 15	11 10	6 5	0
COP1 010001	fmt	0 00000	fs	fd	TRUNC.W 001101	6

6 5 5 5 5 6

Format: TRUNC.W fmt
 TRUNC.W.S fd, fs
 TRUNC.W.D fd, fs

MIPS32
 MIPS32

Purpose: Floating Point Truncate to Word Fixed Point

To convert an FP value to 32-bit fixed point, rounding toward zero

Description: FPR[fd] \leftarrow convert_and_round(FPR[fs])

The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format using rounding toward zero (rounding mode 1). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Invalid Operation, Unimplemented Operation

31	26	25	24		6	5	0
COP0 010000	CO 1			Implementation-dependent code		WAIT 100000	

6 1 19 6

Format: WAIT**MIPS32****Purpose:** Enter Standby Mode

Wait for Event

Description:

The WAIT instruction performs an implementation-dependent operation, usually involving a lower power mode. Software may use the code bits of the instruction to communicate additional information to the processor, and the processor may use this information as control for the lower power mode. A value of zero for code bits is the default and must be valid in all implementations.

The WAIT instruction is typically implemented by stalling the pipeline at the completion of the instruction and entering a lower power mode. The pipeline is restarted when an external event, such as an interrupt or external request occurs, and execution continues with the instruction following the WAIT instruction. It is implementation-dependent whether the pipeline restarts when a non-enabled interrupt is requested. In this case, software must poll for the cause of the restart. The assertion of any reset or NMI must restart the pipeline and the corresponding exception must be taken.

If the pipeline restarts as the result of an enabled interrupt, that interrupt is taken between the WAIT instruction and the following instruction (EPC for the interrupt points at the instruction following the WAIT instruction).

Restrictions:

The operation of the processor is **UNDEFINED** if a WAIT instruction is placed in the delay slot of a branch or a jump.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
I: Enter implementation dependent lower power mode
I+1:/* Potential interrupt taken here */
```

Exceptions:

Coprocessor Unusable Exception

31	26 25	21 20	16 15	11 10	0
COP0 0100 00	WRPGPR 01 110	rt	rd	0 000 0000 0000	11

Format: WRPGPR rd, rt**MIPS32 Release 2****Purpose:** Write to GPR in Previous Shadow Set

To move the contents of a current GPR to a GPR in the previous shadow set.

Description: SGPR[SRSCtl_{PSS}, rd] ← GPR[rt]The contents of the current GPR *rt* is moved to the shadow GPR register specified by SRSCtl_{PSS} (signifying the previous shadow set number) and *rd* (specifying the register number within that set).**Restrictions:**

In implementations prior to Release 2 of the Architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:SGPR[SRSCtl_{PSS}, rd] ← GPR[rt]**Exceptions:**

Coprocessor Unusable

Reserved Instruction

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL3 011111	0 00000	rt	rd	WSBH 00010	BSHFL 100000	6

Format: WSBH rd, rt**MIPS32 Release 2****Purpose:** Word Swap Bytes Within HalfwordsTo swap the bytes within each halfword of GPR *rt* and store the value into GPR *rd*.**Description:** $\text{GPR}[rd] \leftarrow \text{SwapBytesWithinHalfwords}(\text{GPR}[rt])$ Within each halfword of the lower word of GPR *rt* the bytes are swapped, the result is sign-extended, and stored in GPR *rd*.**Restrictions:**

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

If GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.**Operation:**

```

if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
GPR[rd] ← sign_extend(GPR[rt]23..16 || GPR[rt]31..24 || GPR[rt]7..0 || GPR[rt]15..8)

```

Exceptions:

Reserved Instruction

Programming Notes:

The WSBH instruction can be used to convert halfword and word data of one endianness to another endianness. The endianness of a word value can be converted using the following sequence:

```

lw      t0, 0(a1)          /* Read word value */
wsbh   t0, t0              /* Convert endianness of the halfwords */
rotr   t0, t0, 16          /* Swap the halfwords within the words */

```

Combined with SEH and SRA, two contiguous halfwords can be loaded from memory, have their endianness converted, and be sign-extended into two word values in four instructions. For example:

```

lw      t0, 0(a1)          /* Read two contiguous halfwords */
wsbh   t0, t0              /* Convert endianness of the halfwords */
seh    t1, t0              /* t1 = lower halfword sign-extended to word */
sra    t0, t0, 16          /* t0 = upper halfword sign-extended to word */

```

Zero-extended words can be created by changing the SEH and SRA instructions to ANDI and SRL instructions, respectively.

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	XOR 100110	6

Format: XOR rd, rs, rt

MIPS32

Purpose: Exclusive OR

To do a bitwise logical Exclusive OR

Description: GPR[rd] \leftarrow GPR[rs] XOR GPR[rt]

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical Exclusive OR operation and place the result into GPR *rd*.

Restrictions:

None

Operation:

GPR[rd] \leftarrow GPR[rs] xor GPR[rt]

Exceptions:

None

31	26 25	21 20	16 15	0
XORI 001110	rs	rt	immediate	
6	5	5	16	

Format: XORI rt, rs, immediate**MIPS32****Purpose:** Exclusive OR Immediate

To do a bitwise logical Exclusive OR with a constant

Description: GPR[rt] \leftarrow GPR[rs] XOR immediateCombine 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] xor zero_extend(immediate)**Exceptions:**

None

Instruction Bit Encodings

A.1 Instruction Encodings and Instruction Classes

Instruction encodings are presented in this section; field names are printed here and throughout the book in *italics*.

When encoding an instruction, the primary *opcode* field is encoded first. Most *opcode* values completely specify an instruction that has an *immediate* value or offset.

Opcode values that do not specify an instruction instead specify an instruction class. Instructions within a class are further specified by values in other fields. For instance, *opcode* REGIMM specifies the *immediate* instruction class, which includes conditional branch and trap *immediate* instructions.

A.2 Instruction Bit Encoding Tables

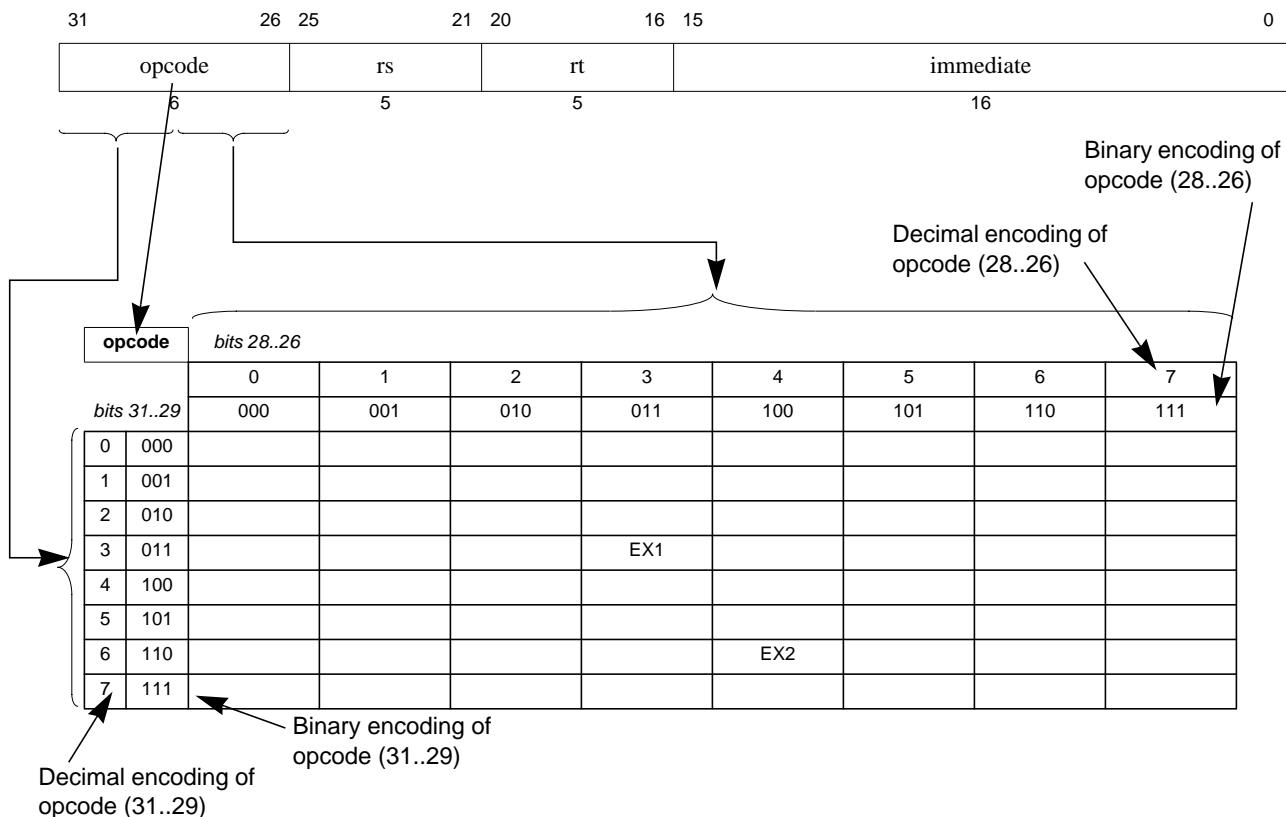
This section provides various bit encoding tables for the instructions of the MIPS64® ISA.

Figure A.1 shows a sample encoding table and the instruction *opcode* field this table encodes. Bits 31..29 of the *opcode* field are listed in the leftmost columns of the table. Bits 28..26 of the *opcode* field are listed along the topmost rows of the table. Both decimal and binary values are given, with the first three bits designating the row, and the last three bits designating the column.

An instruction's encoding is found at the intersection of a row (bits 31..29) and column (bits 28..26) value. For instance, the *opcode* value for the instruction labelled EX1 is 33 (decimal, row and column), or 011011 (binary). Similarly, the *opcode* value for EX2 is 64 (decimal), or 110100 (binary).

Instruction Bit Encodings

Figure A.1 Sample Bit Encoding Table



Tables A.2 through A.23 describe the encoding used for the MIPS64 ISA. Table A.1 describes the meaning of the symbols used in the tables.

Table A.1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction must cause a Reserved Instruction Exception.
δ	(Also <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.
β	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level or a new revision of the Architecture. Executing such an instruction must cause a Reserved Instruction Exception.
\perp	Operation or field codes marked with this symbol represent instructions which are not legal if the processor is configured to be backward compatible with MIPS32 processors. If the processor is executing with 64-bit operations enabled, execution proceeds normally. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).

Table A.1 Symbols Used in the Instruction Encoding Tables (Continued)

Symbol	Meaning
▽	Operation or field codes marked with this symbol represent instructions which were only legal if 64-bit operations were enabled on implementations of Release 1 of the Architecture. In Release 2 of the architecture, operation or field codes marked with this symbol represent instructions which are legal if 64-bit floating point operations are enabled. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
θ	Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, MIPS Technologies will assist the partner in selecting appropriate encodings if requested by the partner. The partner is not required to consult with MIPS Technologies when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (<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).
σ	Field codes marked with this symbol represent an EJTAG support instruction and implementation of this encoding is optional for each implementation. If the encoding is not implemented, executing such an instruction must cause a Reserved Instruction Exception. If the encoding is implemented, it must match the instruction encoding as shown in the table.
ε	Operation or field codes marked with this symbol are reserved for MIPS optional Module or Application Specific Extensions. If the Module/ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception.
∅	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS64 ISA. Software should avoid using these operation or field codes.
⊕	Operation or field codes marked with this symbol are valid for Release 2 implementations of the architecture. Executing such an instruction in a Release 1 implementation must cause a Reserved Instruction Exception.

Table A.2 MIPS64 Encoding of the Opcode Field

opcode		bits 28..26							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	<i>SPECIAL</i> δ	<i>REGIMM</i> δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	<i>COP0</i> δ	<i>COP1</i> δ	<i>COP2</i> θδ	<i>COP1X</i> δ	BEQL φ	BNEL φ	BLEZL φ	BGTZL φ
3	011	DADDI ⊥	DADDIU ⊥	LDL ⊥	LDR ⊥	<i>SPECIAL2</i> δ	JALX ε	MSA εδ	<i>SPECIAL3¹</i> δ⊕
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU ⊥
5	101	SB	SH	SWL	SW	SDL ⊥	SDR ⊥	SWR	CACHE
6	110	LL	LWC1	LWC2 θ	PREF	LLD ⊥	LDC1	LDC2 θ	LD ⊥
7	111	SC	SWC1	SWC2 θ	*	SCD ⊥	SDC1	SDC2 θ	SD ⊥

1. Release 2 of the Architecture added the *SPECIAL3* opcode. Implementations of Release 1 of the Architecture signaled a Reserved Instruction Exception for this opcode.

Instruction Bit Encodings

Table A.3 MIPS64 SPECIAL Opcode Encoding of Function Field

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL ¹	MOVCI δ	SRL δ	SRA	SLLV	*	SRLV δ	SRAV
1	001	JR ²	JALR ²	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	DSLLV \perp	*	DSRLV $\delta\perp$	DSRAV \perp
3	011	MULT	MULTU	DIV	DIVU	DMULT \perp	DMULTU \perp	DDIV \perp	DDIVU \perp
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	DADD \perp	DADDU \perp	DSUB \perp	DSUBU \perp
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	DSLL \perp	*	DSRL $\delta\perp$	DSRA \perp	DSLL32 \perp	*	DSRL32 $\delta\perp$	DSRA32 \perp

- Specific encodings of the *rt*, *rd*, and *sa* fields are used to distinguish among the SLL, NOP, SSNOP, EHB and PAUSE functions.
- Specific encodings of the *hint* field are used to distinguish JR from JR.HB and JALR from JALR.HB

Table A.4 MIPS64 REGIMM Encoding of *rt* Field

rt		bits 18..16							
		0	1	2	3	4	5	6	7
bits 20..19		000	001	010	011	100	101	110	111
0	00	BLTZ	BGEZ	BLTZL ϕ	BGEZL ϕ	*	*	*	ϵ
1	01	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	10	BLTZAL	BGEZAL	BLTZALL ϕ	BGEZALL ϕ	*	*	*	*
3	11	*	*	*	*	ϵ	ϵ	*	SYNCI \oplus

Table A.5 MIPS64 SPECIAL2 Encoding of Function Field

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	θ	MSUB	MSUBU	θ	θ
1	001	ϵ	θ	θ	θ	θ	θ	θ	θ
2	010	θ	θ	θ	θ	θ	θ	θ	θ
3	011	θ	θ	θ	θ	θ	θ	θ	θ
4	100	CLZ	CLO	θ	θ	DCLZ \perp	DCLO \perp	θ	θ
5	101	θ	θ	θ	θ	θ	θ	θ	θ
6	110	θ	θ	θ	θ	θ	θ	θ	θ
7	111	θ	θ	θ	θ	θ	θ	θ	SDBBP σ

Table A.6 MIPS64 SPECIAL3¹ Encoding of Function Field for Release 2 of the Architecture

function	bits 2..0							
	0	1	2	3	4	5	6	7
bits 5..3	000	001	010	011	100	101	110	111
0 000	EXT \oplus	DEXTM $\perp\oplus$	DEXTU $\perp\oplus$	DEXT $\perp\oplus$	INS \oplus	DINSM $\perp\oplus$	DINSU $\perp\oplus$	DINS $\perp\oplus$
1 001	ϵ	ϵ	ϵ	*	ϵ	ϵ	*	*
2 010	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
3 011	ϵ	LWLE	LWRE	CACHEE	SBE	SHE	SCE	SWE
4 100	BSHFL $\oplus\delta$	SWLE	SWRE	PREFE	DBSHFL $\perp\oplus\delta$	*	*	*
5 101	LBUE	LHUE	*	*	LBE	LHE	LLE	LWE
6 110	ϵ	ϵ	*	*	ϵ	*	*	*
7 111	ϵ	*	*	RDHWR \oplus	ϵ	*	*	*

1. Release 2 of the Architecture added the SPECIAL3 opcode. Implementations of Release 1 of the Architecture signaled a Reserved Instruction Exception for this opcode and all function field values shown above.

Table A.7 MIPS64 MOVC_I Encoding of tf Bit

tf	bit 16	
	0	1
	MOVF	MOVT

Table A.8 MIPS64¹ SRL Encoding of Shift/Rotate

R	bit 21	
	0	1
	SRL	ROTR

1. Release 2 of the Architecture added the ROTR instruction. Implementations of Release 1 of the Architecture ignored bit 21 and treated the instruction as an SRL

Table A.9 MIPS64¹ SRLV Encoding of Shift/Rotate

R	bit 6	
	0	1
	SRLV	ROTRV

1. Release 2 of the Architecture added the ROTRV instruction. Implementations of Release 1 of the Architecture ignored bit 6 and treated the instruction as an SRLV

Instruction Bit Encodings

Table A.10 MIPS64¹ DSRLV Encoding of Shift/Rotate

R	bit 6	
	0	1
	DSRLV	DROTRV

1. Release 2 of the Architecture added the DROTRV instruction. Implementations of Release 1 of the Architecture ignored bit 6 and treated the instruction as a DSRLV

Table A.11 MIPS64¹ DSRL Encoding of Shift/Rotate

R	bit 21	
	0	1
	DSRL	DROTR

1. Release 2 of the Architecture added the DROTR instruction. Implementations of Release 1 of the Architecture ignored bit 21 and treated the instruction as a DSRL

Table A.12 MIPS64¹ DSRL32 Encoding of Shift/Rotate

R	bit 21	
	0	1
	DSRL32	DROTR32

1. Release 2 of the Architecture added the DROTR32 instruction. Implementations of Release 1 of the Architecture ignored bit 21 and treated the instruction as a DSRL32

Table A.13 MIPS64 BSHFL and DBSHFL Encoding of sa Field¹

sa		bits 8..6							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00			WSBH (BSHFL) DSBH (DBSHFL)				DSHD (DBSHFL)	
1	01								
2	10	SEB (BSHFL)							
3	11	SEH (BSHFL)							

1. The *sa* field is sparsely decoded to identify the final instructions. Entries in this table with no mnemonic are reserved for future use by MIPS Technologies and may or may not cause a Reserved Instruction exception.

Table A.14 MIPS64 COP0 Encoding of rs Field

rs		bits 23..21							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC0	DMFC0 ⊥	*	ε	MTC0	DMTC0 ⊥	*	*
1	01	ε	*	RDPGPR ⊕	MFMCO ¹ δ⊕	ε	*	WRPGPR ⊕	*
2	10	C0 δ							
3	11	C0 δ							

1. Release 2 of the Architecture added the MFMCO function, which is further decoded as the DI (bit 5 = 0) and EI (bit 5 = 1) instructions.

Table A.15 MIPS64 COP0 Encoding of Function Field When rs=CO

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	TLBR	TLBWI	TLBINV	TLBINVF	*	TLBWR	*
1	001	TLBP	ε	ε	ε	ε	*	ε	*
2	010	ε	*	*	*	*	*	*	*
3	011	ERET	*	*	*	*	*	*	DERET σ
4	100	WAIT	*	*	*	*	*	*	*
5	101	ε	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	ε	*	*	*	*	*	*	*

Table A.16 MIPS64 COP1 Encoding of rs Field

rs		bits 23..21							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC1	DMFC1 ⊥	CFC1	MFHC1 ⊕	MTC1	DMTC1 ⊥	CTC1	MTHC1 ⊕
1	01	BC1 δ	BC1ANY2 δε∇	BC1ANY4 δε∇	*	*	*	*	*
2	10	S δ	D δ	*	*	W δ	L δ	PS δ	*
3	11	*	*	*	*	*	*	*	*

Instruction Bit Encodings

Table A.17 MIPS64 COP1 Encoding of Function Field When $rs=S$

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L ∇	TRUNC.L ∇	CEIL.L ∇	FLOOR.L ∇	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	MOVCF δ	MOVZ	MOVN	*	RECIP ∇	RSQRT ∇	*
3	011	*	*	*	*	RECIP2 $\epsilon\nabla$	RECIP1 $\epsilon\nabla$	RSQRT1 $\epsilon\nabla$	RSQRT2 $\epsilon\nabla$
4	100	*	CVT.D	*	*	CVT.W	CVT.L ∇	CVT.PS ∇	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F CABS.F $\epsilon\nabla$	C.UN CABS.UN $\epsilon\nabla$	C.EQ CABS.EQ $\epsilon\nabla$	C.UEQ CABS.UEQ $\epsilon\nabla$	C.OLT CABS.OLT $\epsilon\nabla$	C.ULT CABS.ULT $\epsilon\nabla$	C.OLE CABS.OLE $\epsilon\nabla$	C.ULE CABS.ULE $\epsilon\nabla$
7	111	C.SF CABS.SF $\epsilon\nabla$	C.NGLE CABS.NGLE $\epsilon\nabla$	C.SEQ CABS.SEQ $\epsilon\nabla$	C.NGL CABS.NGL $\epsilon\nabla$	C.LT CABS.LT $\epsilon\nabla$	C.NGE CABS.NGE $\epsilon\nabla$	C.LE CABS.LE $\epsilon\nabla$	C.NGT CABS.NGT $\epsilon\nabla$

Table A.18 MIPS64 COP1 Encoding of Function Field When $rs=D$

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L ∇	TRUNC.L ∇	CEIL.L ∇	FLOOR.L ∇	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	MOVCF δ	MOVZ	MOVN	*	RECIP ∇	RSQRT ∇	*
3	011	*	*	*	*	RECIP2 $\epsilon\nabla$	RECIP1 $\epsilon\nabla$	RSQRT1 $\epsilon\nabla$	RSQRT2 $\epsilon\nabla$
4	100	CVT.S	*	*	*	CVT.W	CVT.L ∇	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F CABS.F $\epsilon\nabla$	C.UN CABS.UN $\epsilon\nabla$	C.EQ CABS.EQ $\epsilon\nabla$	C.UEQ CABS.UEQ $\epsilon\nabla$	C.OLT CABS.OLT $\epsilon\nabla$	C.ULT CABS.ULT $\epsilon\nabla$	C.OLE CABS.OLE $\epsilon\nabla$	C.ULE CABS.ULE $\epsilon\nabla$
7	111	C.SF CABS.SF $\epsilon\nabla$	C.NGLE CABS.NGLE $\epsilon\nabla$	C.SEQ CABS.SEQ $\epsilon\nabla$	C.NGL CABS.NGL $\epsilon\nabla$	C.LT CABS.LT $\epsilon\nabla$	C.NGE CABS.NGE $\epsilon\nabla$	C.LE CABS.LE $\epsilon\nabla$	C.NGT CABS.NGT $\epsilon\nabla$

Table A.19 MIPS64 COP1 Encoding of Function Field When $rs=W$ or L^1

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	*	*	*	*	*	*	*
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	CVT.S	CVT.D	*	*	*	*	CVT.PS.PW $\epsilon\nabla$	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

1. Format type L is legal only if 64-bit floating point operations are enabled.

Table A.20 MIPS64 COP1 Encoding of Function Field When *rs=PS*¹

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD ∇	SUB ∇	MUL ∇	*	*	ABS ∇	MOV ∇	NEG ∇
1	001	*	*	*	*	*	*	*	*
2	010	*	MOVCF $\delta\nabla$	MOVZ ∇	MOVN ∇	*	*	*	*
3	011	ADDR $\epsilon\nabla$	*	MULR $\epsilon\nabla$	*	RECIP2 $\epsilon\nabla$	RECIP1 $\epsilon\nabla$	RSQRT1 $\epsilon\nabla$	RSQRT2 $\epsilon\nabla$
4	100	CVT.S.PU ∇	*	*	*	CVT.PW.PS $\epsilon\nabla$	*	*	*
5	101	CVT.S.PL ∇	*	*	*	PLL.PS ∇	PLU.PS ∇	PUL.PS ∇	PUU.PS ∇
6	110	C.F ∇ CABS.F $\epsilon\nabla$	C.UN ∇ CABS.UN $\epsilon\nabla$	C.EQ ∇ CABS.EQ $\epsilon\nabla$	C.UEQ ∇ CABS.UEQ $\epsilon\nabla$	C.OLT ∇ CABS.OLT $\epsilon\nabla$	C.ULT ∇ CABS.ULT $\epsilon\nabla$	C.OLE ∇ CABS.OLE $\epsilon\nabla$	C.ULE ∇ CABS.ULE $\epsilon\nabla$
7	111	C.SF ∇ CABS.SF $\epsilon\nabla$	C.NGLE ∇ CABS.NGLE $\epsilon\nabla$	C.SEQ ∇ CABS.SEQ $\epsilon\nabla$	C.NGL ∇ CABS.NGL $\epsilon\nabla$	C.LT ∇ CABS.LT $\epsilon\nabla$	C.NGE ∇ CABS.NGE $\epsilon\nabla$	C.LE ∇ CABS.LE $\epsilon\nabla$	C.NGT ∇ CABS.NGT $\epsilon\nabla$

1. Format type *PS* is legal only if 64-bit floating point operations are enabled.

Table A.21 MIPS64 COP1 Encoding of *tf* Bit When *rs=S, D, or PS*, Function=*MOVCF*

tf	bit 16	
	0	1
MOVF.fmt	MOVTF.fmt	

Table A.22 MIPS64 COP2 Encoding of *rs* Field

rs		bits 23..21							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC2 θ	DMFC2 $\theta\perp$	CFC2 θ	MFHC2 $\theta\oplus$	MTC2 θ	DMTC2 $\theta\perp$	CTC2 θ	MTHC2 $\theta\oplus$
1	01	BC2 θ	*	*	*	*	*	*	*
2	10								
3	11	C2 $\theta\delta$							

Table A.23 MIPS64 COP1X Encoding of Function Field¹

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	LWXCI ∇	LDXC1 ∇	*	*	*	LUXC1 ∇	*	*
1	001	SWXC1 ∇	SDXC1 ∇	*	*	*	SUXC1 ∇	*	PREFIX ∇
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	ALNV.PS ∇	*
4	100	MADD.S ∇	MADD.D ∇	*	*	*	*	MADD.PS ∇	*
5	101	MSUB.S ∇	MSUB.D ∇	*	*	*	*	MSUB.PS ∇	*
6	110	NMADD.S ∇	NMADD.D ∇	*	*	*	*	NMADD.PS ∇	*
7	111	NMSUB.S ∇	NMSUB.D ∇	*	*	*	*	NMSUB.PS ∇	*

Instruction Bit Encodings

1. COP1X instructions are legal only if 64-bit floating point operations are enabled.

A.3 Floating Point Unit Instruction Format Encodings

Instruction format encodings for the floating point unit are presented in this section. This information is a tabular presentation of the encodings described in tables [Table A.16](#) and [Table A.23](#) above.

Table A.24 Floating Point Unit Instruction Format Encodings

<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	Internal review copy of reorganized and updated architecture documentation.
0.92	December 15, 2000	Changes in this revision: <ul style="list-style-type: none"> • Correct sign in description of MSUBU. • Update JR and JALR instructions to reflect the changes required by MIPS16.
0.95	March 12, 2001	Update for second external review release
1.00	August 29, 2002	Update based on all review feedback: <ul style="list-style-type: none"> • Add missing optional select field syntax in mtc0/mfc0 instruction descriptions. • Correct the PREF instruction description to acknowledge that the PrepareForStore function does, in fact, modify architectural state. • To provide additional flexibility for Coprocessor 2 implementations, extend the <i>se</i>/field for DMFC0, DMTC0, MFC0, and MTC0 to be 8 bits. • Update the PREF instruction to note that it may not update the state of a locked cache line. • Remove obviously incorrect documentation in DIV and DIVU with regard to putting smaller numbers in register <i>rt</i>. • Fix the description for MFC2 to reflect data movement from the coprocessor 2 register to the GPR, rather than the other way around. • Correct the pseudo code for LDC1, LDC2, SDC1, and SDC2 for a MIPS32 implementation to show the required word swapping. • Indicate that the operation of the CACHE instruction is UNPREDICTABLE if the cache line containing the instruction is the target of an invalidate or writeback invalidate. • Indicate that an Index Load Tag or Index Store Tag operation of the CACHE instruction must not cause a cache error exception. • Make the entire right half of the MFC2, MTC2, CFC2, CTC2, DMFC2, and DMTC2 instructions implementation dependent, thereby acknowledging that these fields can be used in any way by a Coprocessor 2 implementation. • Clean up the definitions of LL, SC, LLD, and SCD. • Add a warning that software should not use non-zero values of the stype field of the SYNC instruction. • Update the compatibility and subsetting rules to capture the current requirements.

Revision History

Revision	Date	Description
1.90	September 1, 2002	<p>Merge the MIPS Architecture Release 2 changes in for the first release of a Release 2 processor. Changes in this revision include:</p> <ul style="list-style-type: none"> • All new Release 2 instructions have been included: DEXT, DEXTM, DEXTU, DI, DINS, DINSM, DINSU, DROTR, DROTR32, DROTRV, DSBH, DSHD, EHB, EI, EXT, INS, JALR.HB, JR.HB, MFHC1, MFHC2, MTHC1, MTHC2, RDHWR, RDPGPR, ROTR, ROTRV, SEB, SEH, SYNCI, WRPGPR, WSBH. • The following instruction definitions changed to reflect Release 2 of the Architecture: DERET, DSRL, DSRL32, DSRLV, ERET, JAL, JALR, JR, SRL, SRLV • With support for 64-bit FPUs on 32-bit CPUs in Release 2, all floating point instructions that were previously implemented by MIPS64 processors have been modified to reflect support on either MIPS32 or MIPS64 processors in Release 2. • All pseudo-code functions have been updated, and the Are64bitFPOperationsEnabled function was added. • Update the instruction encoding tables for Release 2.
2.00	June 9, 2003	<p>Continue with updates to merge Release 2 changes into the document.</p> <p>Changes in this revision include:</p> <ul style="list-style-type: none"> • Correct the target GPR (from rd to rt) in the SLTI and SLTIU instructions. This appears to be a day-one bug. • Correct CPR number, and missing data movement in the pseudocode for the MTC0 instruction. • Add note to indicate that the CACHE instruction does not take Address Error Exceptions due to mis-aligned effective addresses. • Update SRL, ROTR, SRLV, ROTRV, DSRL, DROTR, DSRLV, DROTRV, DSRL32, and DROTR32 instructions to reflect a 1-bit, rather than a 4-bit decode of shift vs. rotate function. • Add programming note to the PrepareForStore PREF hint to indicate that it can not be used alone to create a bzero-like operation. • Add note to the PREF and PREFIX instruction indicating that they may cause Bus Error and Cache Error exceptions, although this is typically limited to systems with high-reliability requirements. • Update the SYNCI instruction to indicate that it should not modify the state of a locked cache line. • Establish specific rules for when multiple TLB matches can be reported (on writes only). This makes software handling easier.

Revision	Date	Description
2.50	July 1, 2005	<p>Changes in this revision:</p> <ul style="list-style-type: none"> • Correct figure label in LWR instruction (it was incorrectly specified as LWL). • Update all files to FrameMaker 7.1. • Include support for implementation-dependent hardware registers via RDHWR. • Indicate that it is implementation-dependent whether prefetch instructions cause EJTAG data breakpoint exceptions on an address match, and suggest that the preferred implementation is not to cause an exception. • Correct the MIPS32 pseudocode for the LDC1, LDXC1, LUXC1, SDC1, SDXC1, and SUXC1 instructions to reflect the Release 2 ability to have a 64-bit FPU on a 32-bit CPU. The correction simplifies the code by using the ValueFPR and StoreFPR functions, which correctly implement the Release 2 access to the FPRs. • Add an explicit recommendation that all cache operations that require an index be done by converting the index to a kseg0 address before performing the cache operation. • Expand on restrictions on the PREF instruction in cases where the effective address has an uncached coherency attribute. •
2.60	June 25, 2008	<p>Changes in this revision:</p> <ul style="list-style-type: none"> • Applied the new B0.01 template. • Update RDHWR description with the UserLocal register. • added PAUSE instruction • Ordering SYNCs • CMP behavior of CACHE, PREF*, SYNCI • DCLZ, DCLO operations was inverted • CVT.S.PL, CVT.S.PU are non-arithmetic (no exceptions) • *MADD(fmt & *MSUB fmt are non-fused. • various typos fixed
2.61	July 10, 2008	<ul style="list-style-type: none"> • Revision History file was incorrectly copied from Volume III. • Removed index conditional text from PAUSE instruction description. • SYNC instruction - added additional format “SYNC stype”
2.62	January 2, 2009	<ul style="list-style-type: none"> • LWC1, LWXC1 - added statement that upper word in 64bit registers are UNDEFINED. • CVT.S.PL and CVT.S.PU descriptions were still incorrectly listing IEEE exceptions. • Typo in CFC1 Description. • CCRes is accessed through \$3 for RDHWR, not \$4.
3.00	March 25, 2010	<ul style="list-style-type: none"> • JALX instruction description added. • Sub-setting rules updated for JALX. •
3.01	June 01, 2010	<ul style="list-style-type: none"> • Copyright page updated. • User mode instructions not allowed to produce UNDEFINED results, only UNPREDICTABLE results.
3.02	March 21, 2011	<ul style="list-style-type: none"> • RECIP, RSQRT instructions do not require 64-bit FPU. • MADD/MSUB/NMADD/NMSUB psuedo-code was incorrect for PS format check.

Revision History

Revision	Date	Description
3.50	September 20, 2012	<ul style="list-style-type: none">Added EVA load/store instructions: LBE, LBUE, LHE, LHUE, LWE, SBE, SHE, SWE, CACHEE, PREFE, LLE, SCE, LWLE, LWRE, SWLE, SWRE.TLBWI - can be used to invalidate the VPN2 field of a TLB entry.FCSR.MAC2008 bit affects intermediate rounding in MADD(fmt, MSUB(fmt, NMADD(fmt and NMSUB(fmt.FCSR.ABS2008 bit defines whether ABS(fmt and NEG(fmt are arithmetic or not (how they deal with QNAN inputs).
3.51	October 20, 2012	<ul style="list-style-type: none">CACHE and SYNCI ignore RI and XI exceptions.CVT, CEIL, FLOOR, ROUND, TRUNC to integer can't generate FP-Overflow exception.
5.00	December 14, 2012	<ul style="list-style-type: none">R5 changes: DSP and MT ASEs -> ModulesNMADD(fmt, NMSUB(fmt - for IEEE2008 negate portion is arithmetic.
5.01	December 15, 2012	<ul style="list-style-type: none">No technical content changes:Update logos on Cover.Update copyright page.