



**MIPS® Architecture for Programmers
Volume II-B: The microMIPS64™
Instruction Set**

Document Number: MD00594

Revision 5.04

January 15, 2014

Table of Contents

Chapter 1: About This Book	18
1.1: Typographical Conventions	18
1.1.1: Italic Text	19
1.1.2: Bold Text	19
1.1.3: Courier Text	19
1.2: UNPREDICTABLE and UNDEFINED	19
1.2.1: UNPREDICTABLE	19
1.2.2: UNDEFINED	20
1.2.3: UNSTABLE	20
1.3: Special Symbols in Pseudocode Notation	20
1.4: For More Information	23
Chapter 2: Guide to the Instruction Set	24
2.1: Understanding the Instruction Fields	24
2.1.1: Instruction Fields	25
2.1.2: Instruction Descriptive Name and Mnemonic	26
2.1.3: Format Field	26
2.1.4: Purpose Field	27
2.1.5: Description Field	27
2.1.6: Restrictions Field	27
2.1.7: Operation Field	28
2.1.8: Exceptions Field	28
2.1.9: Programming Notes and Implementation Notes Fields	29
2.2: Operation Section Notation and Functions	29
2.2.1: Instruction Execution Ordering	29
2.2.2: Pseudocode Functions	29
2.2.2.1: Coprocessor General Register Access Functions	29
2.2.2.2: Memory Operation Functions	31
2.2.2.3: Floating Point Functions	34
2.2.2.4: Miscellaneous Functions	37
2.3: Op and Function Subfield Notation	39
2.4: FPU Instructions	39
Chapter 3: Introduction	41
3.1: Release 3 of the MIPS Architecture	41
3.2: Default ISA Mode	42
3.3: Software Detection	42
3.4: Compliance and Subsetting	42
3.5: ISA Mode Switch	43
3.6: Branch and Jump Offsets	43
3.7: Coprocessor Unusable Behavior	44
Chapter 4: Instruction Formats	45
4.1: Instruction Stream Organization and Endianness	48
Chapter 5: microMIPS Re-encoded Instructions	51

5.1: 16-Bit Category.....	51
5.1.1: Frequent MIPS64 Instructions.....	51
5.1.2: Frequent MIPS64 Instruction Sequences	54
5.1.3: Instruction-Specific Register Specifiers and Immediate Field Encodings	55
5.2: 16-bit Instruction Register Set	56
5.3: 32-Bit Category.....	58
5.3.1: New 32-bit instructions.....	58
5.4: New Instructions	61
ADDIUPC	62
ADDIUR1SP	64
ADDIUR2.....	66
ADDIUSP	68
ADDIUS5.....	70
ADDU16	72
ANDI16.....	74
AND16.....	76
B16.....	78
BEQZ16.....	80
BEQZC	82
BGEZALS	84
BLTZALS	86
BNEZ16.....	88
BNEZC	90
BREAK16	92
JALR16.....	94
JALRS16	96
JALRS	98
JALRS.HB	100
JALS	104
JALX.....	106
JR16.....	108
JRADDIUSP	110
JRC	112
LBU16	114
LDM	116
LDP	118
LHU16	120
LI16	122
LW16.....	124
LWM32.....	126
LWM16.....	128
LWP.....	130
LWGP.....	132
LWSP	134
LWXS	136
MFHI16.....	138
MFLO16	140
MOVE16.....	142
MOVEP	144
NOT16.....	146
OR16	148
SB16.....	150
SDBBP16	152

SDM	154
SDP	156
SH16	158
SLL16	160
SRL16	162
SUBU16	164
SW16	166
SWSP	168
SWM32	170
SWM16	172
SWP	174
XOR16	176
5.5: Recoded 32-Bit Instructions	177
ABS.fmt	178
ADD	180
ADD.fmt	181
ADDI	183
ADDIU	184
ADDU	185
ALNV.PS	186
AND	189
ANDI	190
B	191
BAL	192
BC1F	194
BC1T	196
BC2F	198
BC2T	200
BEQ	202
BGEZ	203
BGEZAL	204
BGTZ	205
BLEZ	206
BLTZ	207
BLTZAL	208
BNE	209
BREAK	210
C.cond.fmt	211
CACHE	217
CACHEE	224
CEIL.L.fmt	231
CEIL.W.fmt	233
CFC1	235
CFC2	237
CLO	238
CLZ	239
COP2	240
CTC1	241
CTC2	244
CVT.D.fmt	245
CVT.L.fmt	247
CVT.PS.S	249
CVT.S.fmt	251

CVT.S.PL	253
CVT.S.PU	255
CVT.W.fmt	257
DADD	259
DADDI	260
DADDIU	261
DADDU	262
DCLO	263
DCLZ	264
DDIV	265
DDIVU	266
DERET	267
DEXT	269
DEXTM	271
DEXTU	273
DI	275
DINS	277
DINSM	280
DINSU	283
DIV	285
DIV.fmt	287
DIVU	288
DMFC0	290
DMFC2	292
DMFC1	293
DMTC0	295
DMTC1	296
DMTC2	297
DMULT	298
DMULTU	300
DROTR	302
DROTR32	303
DROTRV	304
DSBH	305
DSHD	307
DSLL	309
DSLL32	310
DSLLV	311
DSRA	312
DSRA32	313
DSRAV	314
DSRL	315
DSRL32	316
DSRLV	317
DSUB	318
DSUBU	319
EHB	321
EI	323
ERET	325
ERETNC	326
EXT	328
FLOOR.L.fmt	330
FLOOR.W.fmt	332

INS	334
J.....	337
JAL	338
JALR.....	340
JALR.HB.....	342
JR	346
JR.HB	348
LB	351
LBE.....	352
LBU	354
LBUE	356
LD.....	358
LDC1	360
LDC2	361
LDL.....	363
LDR	365
LDXC1.....	368
LH.....	370
LHE	372
LHU	374
LHUE.....	376
LL	378
LLD	380
LLE	382
LUI.....	384
LUXC1.....	385
LW	387
LWC1	388
LWC2	389
LWE.....	390
LWL.....	392
LWLE.....	394
LWR	396
LWRE	400
LWU	404
LWXC1.....	406
MADD.....	408
MADD.fmt.....	409
MADDDU	411
MFC0.....	412
MFC1.....	413
MFC2.....	414
MFHC0	415
MFHC1	417
MFHC2	418
MFHI.....	419
MFLO	420
MOV.fmt	421
MOVF	423
MOVF.fmt	424
MOVN.....	426
MOVN.fmt.....	427
MOVNT	429

MOVT.fmt	430
MOVZ	432
MOVZ.fmt	433
MSUB	435
MSUB.fmt	436
MSUBU	438
MTC0	439
MTC1	441
MTC2	443
MTHC0	444
MTHC1	446
MTHC2	447
MTHI	449
MTLO	450
MUL	451
MUL.fmt	453
MULT	455
MULTU	457
NEG.fmt	459
NMADD.fmt	461
NMSUB.fmt	463
NOP	465
NOR	466
OR	467
ORI	468
PAUSE	470
PLL.PS	472
PLU.PS	473
PREF	474
PREFE	478
PREFX	481
PUL.PS	483
PUU.PS	484
RDHWR	485
RDPGPR	488
RECIP.fmt	489
ROTR	491
ROTRV	492
ROUND.L.fmt	493
ROUND.W.fmt	495
RSQRT.fmt	497
SB	499
SBE	500
SC	502
SCE	506
SCD	509
SD	512
SDBBP	513
SDC1	514
SDC2	515
SDL	516
SDR	518
SDXC1	520

SEB	522
SEH	524
SH	526
SHE	528
SLL	530
SLLV	531
SLT	532
SLTI	533
SLTIU	534
SLTU	535
SQRT.fmt	536
SRA	538
SRAV	539
SRL	540
SRLV	541
SSNOP	542
SUB	543
SUB.fmt	544
SUBU	546
SUXC1	547
SW	548
SWE	550
SWC1	551
SWC2	552
SWL	554
SWLE	556
SWR	559
SWRE	562
SWXC1	565
SYNC	567
SYNCI	573
SYSCALL	576
TEQ	577
TEQI	578
TGE	579
TGEI	580
TGEIU	581
TGEU	582
TLBP	584
TLBR	586
TLBWI	588
TLBWR	590
TLT	592
TLTI	593
TLTIU	594
TLTU	595
TNE	596
TNEI	597
TRUNC.L.fmt	598
TRUNC.W.fmt	600
WAIT	602
WRPGPR	604
WSBH	605

XOR.....	607
XORI.....	608
Chapter 7: Opcode Map	609
7.1: Major Opcodes	609
7.2: Minor Opcodes	611
7.3: Floating Point Unit Instruction Format Encodings	620
Chapter 8: Compatibility	623
8.1: Assembly-Level Compatibility.....	623
8.2: ABI Compatibility	624
8.3: Branch and Jump Offsets	625
8.4: Relocation Types.....	625
8.5: Boot-up Code shared between microMIPS64 and MIPS64	625
8.6: Coprocessor Unusable Behavior	626
8.7: Other Issues Affecting Software and Compatibility	626
Appendix C: References	627
Appendix D: Revision History	629

List of Tables

Table 1.1: Symbols Used in Instruction Operation Statements.....	20
Table 2.1: AccessLength Specifications for Loads/Stores.....	33
Table 4.1: microMIPS Opcode Formats.....	48
Table 5.1: 16-Bit Re-encoding of Frequent MIPS64 Instructions.....	52
Table 5.2: 16-Bit Re-encoding of Frequent MIPS64 Instruction Sequences.....	54
Table 5.3: Instruction-Specific Register Specifiers and Immediate Field Values.....	55
Table 5.4: 16-Bit Instruction General-Purpose Registers - \$2-\$7, \$16, \$17.....	56
Table 5.5: SB16, SH16, SW16 Source Registers - \$0, \$2-\$7, \$17.....	57
Table 5.6: 16-Bit Instruction Implicit General-Purpose Registers.....	58
Table 5.7: 16-Bit Instruction Special-Purpose Registers.....	58
Table 5.8: 32-bit Instructions introduced within microMIPS.....	58
Table 5.9: Encoded and Decoded Values of the Immediate Field.....	66
Table 5.10: Encoded and Decoded Values of Immediate Field.....	68
Table 5-1: Encoded and Decoded Values of Signed Immediate Field.....	70
Table 5-2: Encoded and Decoded Values of Immediate Field.....	74
Table 5.11: Offset Field Encoding Range -1, 0..14.....	114
Table 5.12: LI16 -1, 0..126 Immediate Field Encoding Range.....	122
Table 5.13: Encoded and Decoded Values of the Enc_Dest Field.....	144
Table 5.14: Encoded and Decoded Values of the Enc_rs and Enc_rt Fields.....	144
Table 5.15: Shift Amount Field Encoding.....	160
Table 5.16: Shift Amount Field Encoding.....	162
Table 5.17: FPU Comparisons Without Special Operand Exceptions.....	213
Table 5.18: FPU Comparisons With Special Operand Exceptions for QNaNs.....	214
Table 5.19: Usage of Effective Address.....	217
Table 5.20: Encoding of Bits[17:16] of CACHE Instruction.....	218
Table 5.21: Encoding of Bits [20:18] of the CACHE Instruction.....	219
Table 5.22: Usage of Effective Address.....	224
Table 5.23: Encoding of Bits[22:21] of CACHEE Instruction.....	225
Table 5.24: Encoding of Bits [20:18] of the CACHEE Instruction.....	226
Table 6.25: Values of <i>hint</i> Field for PREF Instruction.....	474
Table 6.26: Values of <i>hint</i> Field for PREFE Instruction.....	479
Table 6.27: RDHWR Register Register Numbers.....	485
Table 6.28: Encodings of the Bits[10:6] of the SYNC instruction; the SType Field.....	569
Table 7.1: Symbols Used in the Instruction Encoding Tables.....	610
Table 7.2: microMIPS64 Encoding of Major Opcode Field.....	611
Table 7.3: Legend for Minor Opcode Tables.....	612
Table 7.4: POOL32A Encoding of Minor Opcode Field.....	612
Table 7.5: POOL32Axf Encoding of Minor Opcode Extension Field.....	613
Table 7.6: POOL32F Encoding of Minor Opcode Field.....	614
Table 7.7: POOL32Fxf Encoding of Minor Opcode Extension Field.....	615
Table 7.8: POOL32B Encoding of Minor Opcode Field.....	615
Table 7.9: POOL32C Encoding of Minor Opcode Field.....	616
Table 7.10: LD-EVA Encoding of Minor Opcode Field.....	616
Table 7.11: ST-EVA Encoding of Minor Opcode Field.....	616
Table 7.12: POOL32I Encoding of Minor Opcode Field.....	617
Table 7.13: POOL32S Encoding of Minor Opcode Field.....	617
Table 7.14: POOL32Sxf Encoding of Minor Opcode Extension Field.....	618

Table 7.15: POOL16A Encoding of Minor Opcode Field	619
Table 7.16: POOL16B Encoding of Minor Opcode Field	619
Table 7.17: POOL16C Encoding of Minor Opcode Field	619
Table 7.18: POOL16D Encoding of Minor Opcode Field	620
Table 7.19: POOL16E Encoding of Minor Opcode Field	620
Table 7.20: POOL16F Encoding of Minor Opcode Field.....	620
Table 7.21: Floating Point Unit Format Encodings - S, D, PS.....	621
Table 7.22: Floating Point Unit Format Encodings - S, D 1-bit	621
Table 7.23: Floating Point Unit Instruction Format Encodings - S, D 2-bits	621
Table 7.24: Floating Point Unit Format Encodings - S, W, L.....	622
Table 7.25: Floating Point Unit Format Encodings - D, W, L	622

List of Figures

Figure 2.1: Example of Instruction Description	25
Figure 2.2: Example of Instruction Fields	26
Figure 2.3: Example of Instruction Descriptive Name and Mnemonic	26
Figure 2.4: Example of Instruction Format	26
Figure 2.5: Example of Instruction Purpose	27
Figure 2.6: Example of Instruction Description	27
Figure 2.7: Example of Instruction Restrictions	28
Figure 2.8: Example of Instruction Operation	28
Figure 2.9: Example of Instruction Exception	28
Figure 2.10: Example of Instruction Programming Notes	29
Figure 2.11: COP_LW Pseudocode Function	30
Figure 2.12: COP_LD Pseudocode Function	30
Figure 2.13: COP_SW Pseudocode Function	30
Figure 2.14: COP_SD Pseudocode Function	31
Figure 2.15: CoprocessorOperation Pseudocode Function	31
Figure 2.16: AddressTranslation Pseudocode Function	31
Figure 2.17: LoadMemory Pseudocode Function	32
Figure 2.18: StoreMemory Pseudocode Function	32
Figure 2.19: Prefetch Pseudocode Function	33
Figure 2.20: SyncOperation Pseudocode Function	34
Figure 2.21: ValueFPR Pseudocode Function	34
Figure 2.22: StoreFPR Pseudocode Function	35
Figure 2.23: CheckFPEException Pseudocode Function	36
Figure 2.24: FPConditionCode Pseudocode Function	36
Figure 2.25: SetFPConditionCode Pseudocode Function	36
Figure 2.26: SignalException Pseudocode Function	37
Figure 2.27: SignalDebugBreakpointException Pseudocode Function	37
Figure 2.28: SignalDebugModeBreakpointException Pseudocode Function	37
Figure 2.29: NullifyCurrentInstruction PseudoCode Function	38
Figure 2.30: JumpDelaySlot Pseudocode Function	38
Figure 2.31: NotWordValue Pseudocode Function	38
Figure 2.32: PolyMult Pseudocode Function	38
Figure 4.1: 16-Bit Instruction Formats	46
Figure 4.2: 32-Bit Instruction Formats	47
Figure 4.3: Immediate Fields within 32-Bit Instructions	47
Figure 5.1: Example of an ALNV.PS Operation	186
Figure 5.2: Usage of Address Fields to Select Index and Way	217
Figure 5.3: Usage of Address Fields to Select Index and Way	224
Figure 5.4: Operation of the DEXT Instruction	269
Figure 5.5: Operation of the DEXTM Instruction	271
Figure 5.6: Operation of the DEXTU Instruction	273
Figure 5.7: Operation of the DINS Instruction	277
Figure 5.8: Operation of the DINSM Instruction	280
Figure 5.9: Operation of the DINSU Instruction	283
Figure 5.10: Operation of the EXT Instruction	328
Figure 5.11: Operation of the INS Instruction	334
Figure 5.12: Unaligned Doubleword Load Using LDL and LDR	363

Figure 5.13: Bytes Loaded by LDL Instruction	364
Figure 5.14: Unaligned Doubleword Load Using LDR and LDL.....	365
Figure 5.15: Bytes Loaded by LDR Instruction	366
Figure 5.16: Unaligned Word Load Using LWL and LWR.....	392
Figure 5.17: Bytes Loaded by LWL Instruction	393
Figure 5.18: Unaligned Word Load Using LWLE and LWRE	394
Figure 5.19: Bytes Loaded by LWLE Instruction.....	395
Figure 5.20: Unaligned Word Load Using LWL and LWR.....	396
Figure 5.21: Bytes Loaded by LWR Instruction.....	397
Figure 5.22: Unaligned Word Load Using LWLE and LWRE	401
Figure 5.23: Bytes Loaded by LWRE Instruction	401
Figure 6.24: Unaligned Doubleword Store With SDL and SDR	516
Figure 6.25: Bytes Stored by an SDL Instruction.....	517
Figure 6.26: Unaligned Doubleword Store With SDR and SDL	518
Figure 6.27: Bytes Stored by an SDR Instruction	519
Figure 6.28: Unaligned Word Store Using SWL and SWR	554
Figure 6.29: Bytes Stored by an SWL Instruction	555
Figure 6.30: Unaligned Word Store Using SWLE and SWRE	556
Figure 6.31: Bytes Stored by an SWLE Instruction.....	557
Figure 6.32: Unaligned Word Store Using SWR and SWL	559
Figure 6.33: Bytes Stored by SWR Instruction.....	560
Figure 6.34: Unaligned Word Store Using SWRE and SWLE	562
Figure 6.35: Bytes Stored by SWRE Instruction	563
Figure 7.1: Sample Bit Encoding Table	609

About This Book

The MIPS® Architecture for Programmers Volume II-B: The microMIPS64™ 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

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
$*$, \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
<<	Logical Shift left (shift in zeros at right-hand-side)
>>	Logical Shift right (shift in zeros at left-hand-side)
GPRLen	The length in bits (32 or 64) of the CPU general-purpose registers
$GPR[x]$	CPU general-purpose register x . The content of $GPR[0]$ is always zero. In Release 2 of the Architecture, $GPR[x]$ is a short-hand notation for $SGPR[SRSCtl_{CSS}, x]$.
$SGPR[s,x]$	In Release 2 of the Architecture and subsequent releases, multiple copies of the CPU general-purpose registers may be implemented. $SGPR[s,x]$ refers to GPR set s , register x .
$FPR[x]$	Floating Point operand register x
$FCC[CC]$	Floating Point condition code CC . $FCC[0]$ has the same value as $COC[1]$.
$FPR[x]$	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 <i>Status</i> register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).
ReverseEndian	Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the RE bit of the <i>Status</i> 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 <i>PC</i> during an instruction time. If no value is assigned to <i>PC</i> during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the <i>PC</i> during the instruction time of the instruction in the branch delay slot. 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" data-bbox="597 1251 1265 1398"> <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 MIPS16e or microMIPS instructions</td> </tr> </tbody> </table> <p>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.</p>	Encoding	Meaning	0	The processor is executing 32-bit MIPS instructions	1	The processor is executing MIPS16e or microMIPS instructions
Encoding	Meaning						
0	The processor is executing 32-bit MIPS instructions						
1	The processor is executing MIPS16e 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	<p>Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). It is optional if the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR.</p> <p>microMIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a microMIPS32 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.</p> <p>The value of FP32RegistersMode is computed from the FR bit in the <i>Status</i> register.</p>
InstructionInBranchDelaySlot	<p>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.</p>
SignalException(exception, argument)	<p>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.</p>

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

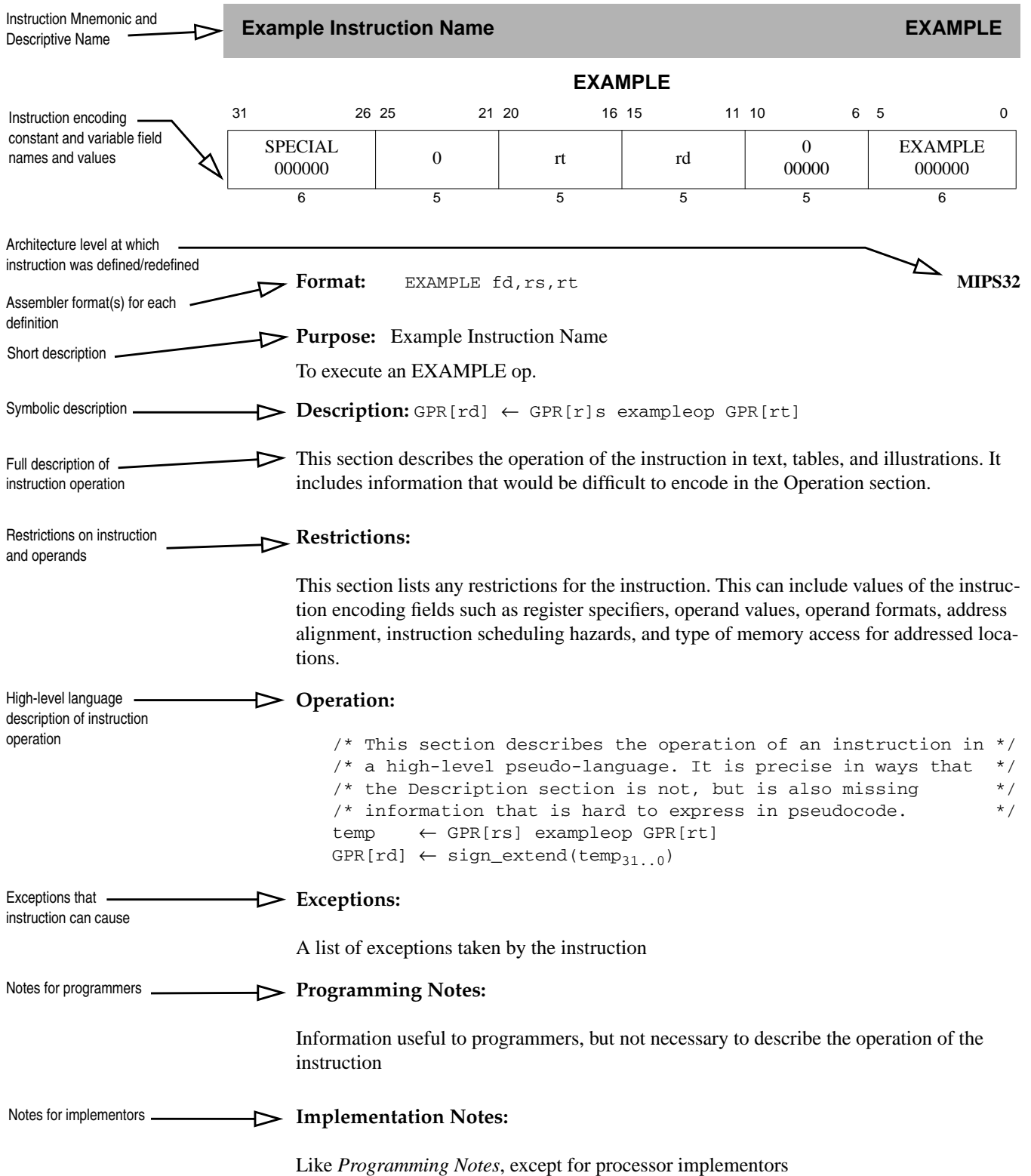
This chapter provides a detailed guide to understanding the instruction descriptions, which are listed in alphabetical order in the tables at the beginning of the next chapter.

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 25
- “Instruction Descriptive Name and Mnemonic” on page 26
- “Format Field” on page 26
- “Purpose Field” on page 27
- “Description Field” on page 27
- “Restrictions Field” on page 27
- “Operation Field” on page 28
- “Exceptions Field” on page 28
- “Programming Notes and Implementation Notes Fields” on page 29

Figure 2.1 Example of Instruction Description

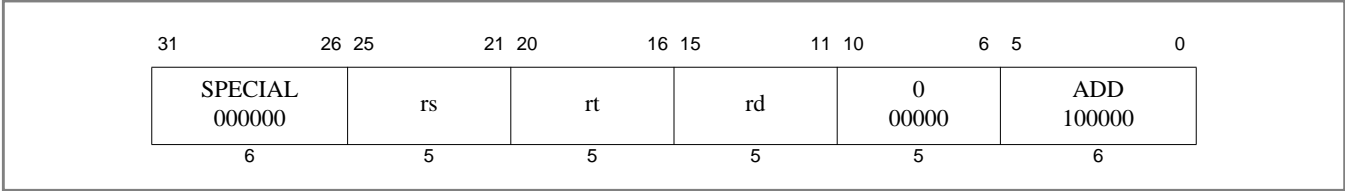


2.1.1 Instruction Fields

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.

The term *decoded_immediate* is used if the immediate field is encoded within the binary format but the assembler format uses the decoded value. The term *left_shifted_offset* is used if the offset field is encoded within the binary format but the assembler format uses value after the appropriate amount of left shifting.

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 29 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 29
- “[Pseudocode Functions](#)” on page 29

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 29
- “[Memory Operation Functions](#)” on page 31
- “[Floating Point Functions](#)” on page 34
- “[Miscellaneous Functions](#)” on page 37

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 in below specifies how the unformatted contents loaded or moved to CPI 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 formattted value from the FPR */

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

case fmt of
    S, W, UNINTERPRETED_WORD:
        valueFPR ← 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
```

```

        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-likely 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 39 for a description of the *op* and *function* subfields.

Introduction

In today's market, the lowest price/performance points must be satisfied, especially for deeply-embedded applications such as microcontroller applications. Moreover, customers require efficient solutions that can be turned into products quickly. To meet this need, the MIPS® instruction set has been optimized and re-encoded into a new variable-length scheme. This solution is called microMIPS™.

microMIPS minimizes the resulting code footprint of applications and it therefore reduces the cost of memory, which is particularly high for embedded memory. Simultaneously, the high performance of MIPS cores is maintained. Using this technology, the customer can generate best results without spending time to profile its application. The smaller code footprint typically leads to reduced power consumption per executed task because of the smaller number of memory accesses.

microMIPS is the preferred replacement for the existing MIPS16e™ ASE. MIPS16e could only be used for user mode programs which did not use floating-point nor any of the Application Specific Extensions (ASEs). microMIPS does not have these limitations - it can be used for kernel mode code as well as user mode programs; it can be used for programs which use floating-point; it can be used with the available ASEs.

microMIPS is also an alternative to the MIPS64® instruction encoding and can be implemented in parallel or stand-alone. The microMIPS equivalent of MIPS32 is named microMIPS32™ and the microMIPS equivalent of MIPS64 is microMIPS64™.

Overview of changes vs. existing MIPS64 ISA:

- 16-bit and 32-bit opcodes
- Optimized opcode/operand field definitions based on statistics
- Branch and jump delay slots will be kept for maximum compatibility and lowest risk
- Removal of branch likely instructions, emulation by assembler
- Fine-tuned register allocation algorithm in the compiler for lowest code size

3.1 Release 3 of the MIPS Architecture

Enhancements included in Release 3 of the MIPS Architecture (also known as MIPSr3™) are:

- microMIPS: The MIPS Release 3 Architecture (also known as MIPSr3™) supports both the MIPS64 instruction set and microMIPS64™ instruction set. Both can be implemented either in parallel or stand-alone. For the first implementations, microMIPS will be primarily implemented together with MIPS64 encoded instruction execution.
- microMIPS is the preferred replacement for MIPS16e. Therefore these two schemes never co-exist within the same processor core.

- Branch likely instructions are phased out in microMIPS and are emulated by the assembler. They remain available in the MIPS64 encoding.

Unless otherwise described in this document, all other aspects of the microMIPS64 architecture are identical to MIPS64 Release 2.

3.2 Default ISA Mode

The instruction sets which are available within an implementation are reported by the *Config3_{ISA}* register field (bits 15:14). *Config1_{CA}* (bit 2) is not used for microMIPS64.

For implementations that support both microMIPS64 and MIPS64, the selected ISA mode following reset is determined by the setting of the *Config3_{ISA}* register field., which is a read-only field set by a hardware signal external to the processor core.

For implementations that support both microMIPS64 and MIPS64, the selected ISA mode upon handling an exception is determined by the setting of the *Config3_{ISAOnExc}* register field (bit 16). The *Config3_{ISAOnExc}* register field is writeable by software and has a reset value that is set by a hardware signal external to the processor core. This register field allows privileged software to change the ISA mode to be used for subsequent exceptions. This capability is for all exception types whose vectors are offsets of the *EBASE* register.

For implementations that support both microMIPS64 and MIPS64, the selected ISA mode when handling a debug exception is determined by the setting of the *ISAOnDebug* register field in the *EJTAG TAP Control* register. This register field is writeable by EJTAG probe software and has a reset value that is set by a hardware signal external to the processor core.

For CPU cores supporting the MT ASE and multiple VPEs, the ISA mode for exceptions can be selected on a per-VPE basis.

3.3 Software Detection

Software can determine if microMIPS64 ISA is implemented by checking the state of the ISA (Instruction Set Architecture) field in the *Config3 CP0* register. *Config1_{CA}* (bit 2) is not used for microMIPS64.

Software can determine if the MIPS64 ISA is implemented by checking the state of the ISA (Instruction Set Architecture) register field in the *Config3 CP0* register.

Software can determine which ISA is used when handling an exception by checking the state of the *ISAOnExc* (ISA on Exception) field in the *Config3 CP0* register.

Debug Probe Software can determine which ISA is used when handling a debug exception by checking the state of the *ISAOnDebug* field in the *EJTAG TAP Control* register.

3.4 Compliance and Subsetting

This document does not change the instruction subsets as defined by the other MIPS architecture reference manuals, including the subsets defined by the various ASEs.

3.5 ISA Mode Switch

The MIPS Release 3 architecture defines an ISA mode for each processor. An ISA mode value of 0 indicates MIPS64 instruction decoding. In processors implementing microMIPS64, an ISA mode value of 1 selects microMIPS64 instruction decoding. In processors implementing the MIPS16e ASE, an ISA mode value of 1 selects the decoding of instructions as MIPS16e.

The ISA mode is not directly visible to user mode software. Upon an exception, the ISA mode of the faulting/interrupted instruction is recorded in the least-significant address bit within the appropriate return address register - either *EPC* or *ErrorEPC* or *DebugEPC*, depending on the exception type.

For the rest of this section, the following definitions are used:

Jump-and-Link-Register instructions: For the MIPS64 ISA, this means the JALR and JALR.HB instructions. For the microMIPS64 ISA, this means the JALR, JALR.HB, JALR16, JALRS, JALRS16 and JALRS.HB instructions.

Jump-Register instructions: For the MIPS64 ISA, this means the JR and JR.HB instructions. For the microMIPS64 ISA, this means the instructions JR, JR.HB, JR16, JRC and JRADDIUSP instructions.

Mode switching between MIPS64 and microMIPS64 uses the same mechanism used by MIPS16e, namely, the JALX, Jump-and-Link-Register and Jump-Register instructions, as described below.

- The JALX instruction executes a JAL and switches to the other mode.
- The Jump-and-Link-Register and Jump-Register instructions interpret bit 0 of the source registers as the target ISA mode (0=MIPS64, 1=microMIPS64) and therefore set the ISA Mode bit according to the contents of bit 0 of the source register. For the actual jump operation, the PC is loaded with the value of the source register with bit 0 set to 0. The Jump-and-Link-Register instructions save the ISA mode into bit 0 of the destination register.
- When exceptions or interrupts occur and the processor writes to *EPC*, *DEPC*, or *ErrorEPC*, the ISA Mode bit is saved into bit 0 of these registers. Then the ISA Mode bit is set according to the *Config3_{ISA}* register field. On return from an exception, the processor loads the ISA Mode bit based on the value from either *EPC*, *DEPC*, or *ErrorEPC*.

If only one ISA mode exists (either MIPS64 or microMIPS64) then this mode switch mechanism does not exist, but the ISA Mode bit is still maintained and has a fixed value (0=MIPS64, 1=microMIPS64). This is to maintain code compatibility between devices which implement both ISA modes and devices which implement only one ISA mode. Executing the JALX instruction will cause a Reserved Instruction exception. Jump-Register and Jump-and-Link-Register instructions cause an Address exception on the target instruction fetch when bit 0 of the source register is different from the fixed ISA mode. Exception handlers must use the instruction set binary format supported by the processor. The Jump-and-Link-Register instructions must still save the fixed ISA mode into bit 0 of the destination register.

3.6 Branch and Jump Offsets

In the MIPS64 architecture, because instructions are always 32 bits in size, the jump and branch target addresses are word (32-bit) aligned. Jump/branch offset fields are shifted left by two bits to create a word-aligned effective address.

In the microMIPS64 architecture, because instructions can be either 16 or 32 bits in size, the jump and branch target addresses are halfword (16-bit) aligned. Branch/jump offset fields are shifted left by only one bit to create halfword-aligned effective addresses.

To maintain the existing MIPS64 ABIs, link unit/object file entry points are restricted to 32-bit word alignments. In the future, a microMIPS64-only ABI can be created to remove this restriction.

3.7 Coprocessor Unusable Behavior

If an instruction associated with a non-implemented coprocessor is executed, it is implementation specific whether a processor executing in microMIPS64 mode raises an RI exception or a coprocessor unusable exception. This behavior is different from the MIPS64 behavior in which coprocessor unusable exception is signalled for such cases.

If the microMIPS64 implementation chooses to use RI exception in such cases, the microMIPS64 RI exception handler must check for coprocessor instructions being executed while the associated coprocessor is implemented but has been disabled (*Status*_{CUs} set to zero).

Instruction Formats

This chapter defines the formats of microMIPS instructions. The microMIPS variable-length encoding comprises 16-bit and 32-bit wide instructions. The 6-bit major opcode is left-aligned within the instruction encoding. Instructions can have 0 to 4 register fields. For 32-bit instructions, the register field width is 5 bits, while for most 16-bit instructions, the register field width is 3 bits, utilizing instruction-specific register encoding. All 5-bit register fields are located at a constant position within the instruction encoding.

The immediate field is right-aligned in the following instructions:

- some 16-bit instructions with 3-bit register fields
- 32-bit instructions with 16-bit or 26-bit immediate field

The name ‘immediate field’ as used here includes the address offset field for branches and load/store instructions as well as the jump target field.

Other instruction-specific fields are typically located between the immediate and minor opcode fields. Instructions that have multiple “other” fields are listed in alphabetical order according to the name of the field, with the first name of the order located at the lower bit position. An empty bit field that is not explicitly shown in the instruction format is located next to the minor opcode field.

[Figure 4.1](#) and [Figure 4.2](#) show the 16-bit and 32-bit instruction formats.

Figure 4.1 16-Bit Instruction Formats

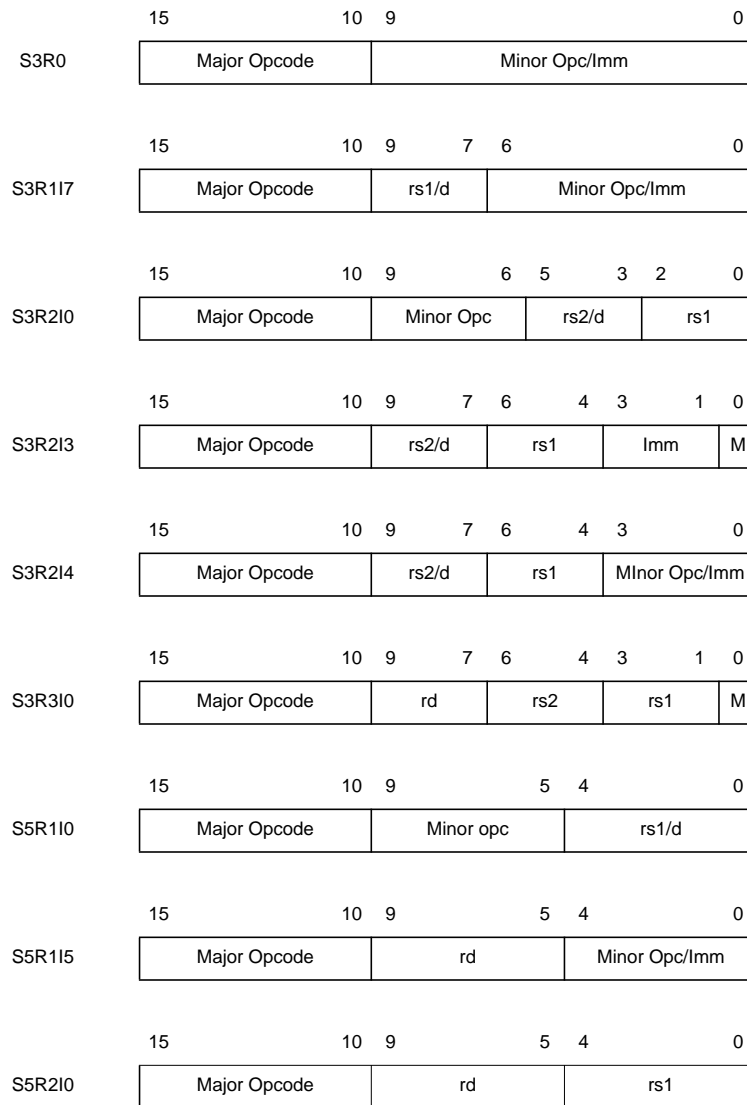


Figure 4.2 32-Bit Instruction Formats

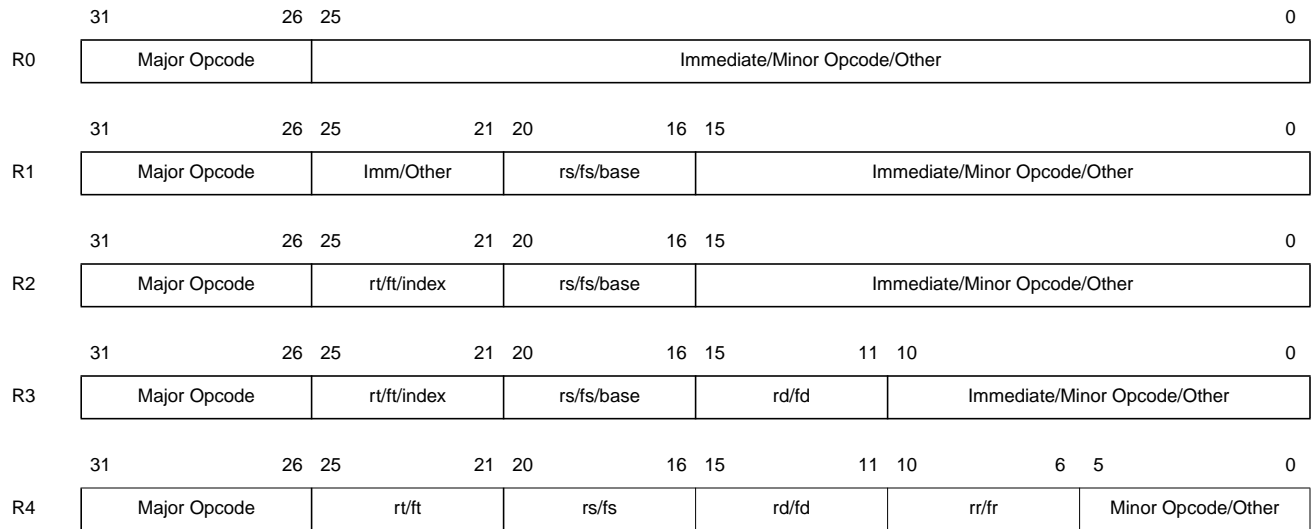
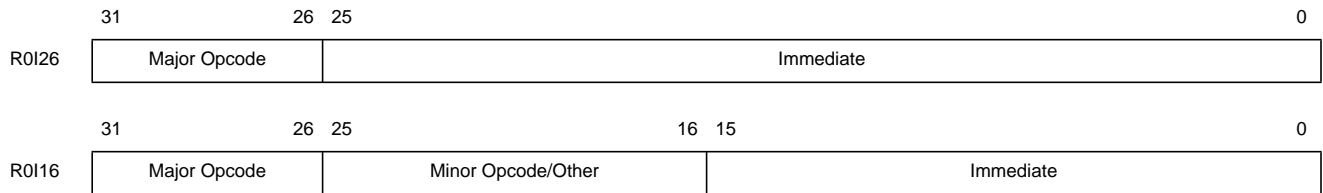
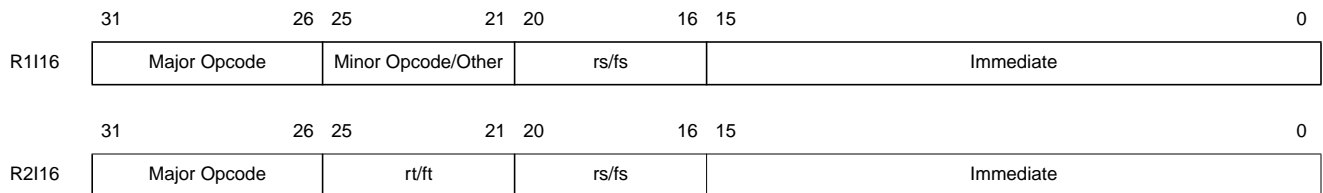


Figure 4.3 Immediate Fields within 32-Bit Instructions

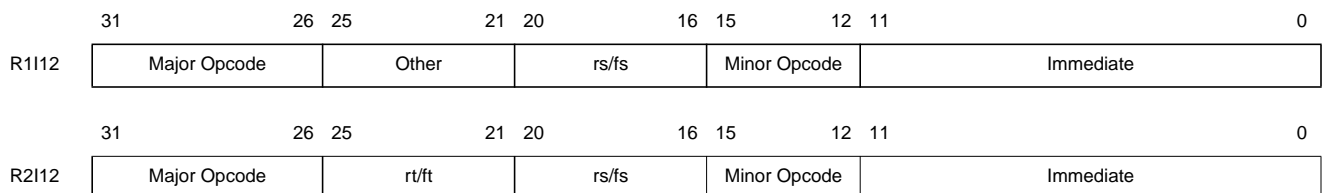
32-bit instruction formats with 26-bit immediate fields:



32-bit instruction formats with 16-bit immediate fields:



32-bit instruction formats with 12-bit immediate fields:



4.1 Instruction Stream Organization and Endianness

The instruction size can be completely derived from the major opcode. For 32-bit instructions, the major opcode also defines the position of the minor opcode field and whether or not the immediate field is right-aligned.

Instructions formats are named according to the number of the register fields and the size of the immediate field. The names have the structure R<x>I<y>. For example, an instruction based on the format R2I16 has 2 register fields and a 16-bit immediate field.

Table 4.1 shows all formats. The 16-bit formats refer to either 3-bit or 5-bit register fields. To visualize this, a 16-bit format name starts with the prefix S3 or S5 respectively.

Table 4.1 microMIPS Opcode Formats

32-bit Instruction Formats (existing instructions)	32-bit Instruction Formats (additional format(s) for new instructions)	16-bit Instruction Formats
R0I0	R2I12	S3R0I0
R0I8		S3R0I10
R0I16		S3R1I7
R0I26		S3R2I0
R1I0		S3R2I3
R1I2		S3R2I4
R1I7		S3R3I1
R1I8		S5R1I0
R1I10		S5R1I4
R1I16		S5R2I0
R2I0		
R2I2		
R2I3		
R2I4		
R2I5		
R2I10		
R2I16		
R3I0		
R3I3		
R4I0		

4.1 Instruction Stream Organization and Endianness

16-bit instructions are placed within the 32-bit (or 64-bit) memory element according to system endianness.

Instruction Formats

- On a 32-bit processor in big-endian mode, the first instruction is read from bits 31..16, and the second instruction is read from bits 15..0.
- On a 32-bit processor in little-endian mode, the first instruction is read from bits 15..0, and the second instruction is read from bits 31..16.

The above rule also applies to the halfwords of 32-bit instructions. This means that a 32-bit instruction is not treated as a word data type; instead, the halfwords are treated in the same way as individual 16-bit instructions. The halfword containing the major opcode is always the first in the sequence.

Example:

```
SRL r1, r1, 7    binary opcode fields: 000000 00001 00001 00111 00001 000000
                  hex representation:  0021 3840

                  Address:   3  2  1  0
Little Endian:    Data:     38 40 00 21

                  Address:   0  1  2  3
Big Endian:      Data:     00 21 38 40
```

Instructions are placed in memory such that they are in-order with respect to the address.

microMIPS Re-encoded Instructions

This chapter lists all microMIPS re-encoded instructions, sorted into 16-bit and 32-bit categories.

In the 16-bit category:

- Frequent MIPS64 instructions and macros, re-encoded as 16-bit. Register and immediate fields are reduced in size by using encodings of frequently occurring values.

In the 32-bit category:

- All MIPS64 instructions, including all application-specific extensions except MIPS16e, re-encoded: MIPS64, MDMX ASE, MIPS-3D ASE, MIPS DSP ASE, MIPS MT ASE, and SmartMIPS ASE.
- Opcode space for user-defined instructions (UDIs).
- New instructions designed primarily to reduce code size.

To differentiate between 16-bit and 32-bit encoded instructions, the instruction mnemonic can be optionally extended with the suffix “16” or “32” respectively. This suffix is placed at the end of the instruction before the first ‘.’ if there is one. For example:

ADD16, ADD32, ADD32.PS, DADDIU32

If these suffixes are omitted, the assembler automatically chooses the smallest instruction size.

For each instruction, the tables in this chapter provide all necessary information about the bit fields. The formats of the instructions are defined in [Chapter 4, “Instruction Formats” on page 45](#). Together with the major and minor opcode encodings, which can be derived from the tables in [Chapter 7, “Opcode Map” on page 609](#), the complete instruction encoding is provided.

Most register fields have a width of 5 bits. 5-bit register fields use linear encoding ($r0=’00000’$, $r1=’00001’$, etc.). For 16-bit instructions, whose register field size is variable, the register field width is explicitly stated in the instruction table ([Table 5.1](#) and [Table 5.2](#)), and the individual register and immediate encodings are shown in [Table 5.3](#). The ‘other fields’ are defined by the respective column, with the order of these fields in the instruction encoding defined by the order in the tables.

5.1 16-Bit Category

5.1.1 Frequent MIPS64 Instructions

These are frequent MIPS64 instructions with reduced register and immediate fields containing frequently used registers and immediate values.

MOVE is a very frequent instruction. It therefore supports full 5-bit unrestricted register fields for maximum efficiency. In fact, MOVE used to be a simplified macro of an existing MIPS64 instruction.

There are 2 variants of the LW and SW instructions. One variant implicitly uses the SP register to allow for a larger offset field. The value in the offset field is shifted left by 2 before it is added to the base address.

There are four variants of the ADDIU instruction:

1. A variant with one 5-bit register specifier that allows any GPR to be the source and destination register
2. A variant that uses the stack pointer as the implicit source and destination register
3. A variant that has separate 3-bit source and destination register specifiers
4. A variant that has the stack pointer as the implicit source register and one 3-bit destination register specifier

A 16-bit NOP instruction is needed because of the new 16-bit instruction alignment and the need in specific cases to align instructions on a 32-bit boundary. It can save code size as well. NOP is not shown in the table because it is realized as a macro (as is NEGU).

```
NOP16 = MOVE16 r0, r0
```

```
NEGU16 rt, rs = SUBU16 rt, r0, rs
```

Because microMIPS instructions are 16-bit aligned, the 16-bit branch instructions support 16-bit aligned branch target addresses. The offset field is left shifted by 1 before it is added to the PC.

The compact instruction JRC is to be used instead of JR, when the jump delay slot after JR cannot be filled. This saves code size. Because JRC may execute as fast as JR with a NOP in the delay slot, JR is preferred if the delay slot can be filled.

The breakpoint instructions, BREAK and SDBBP, include a 16-bit variant that allows a breakpoint to be inserted at any instruction address without overwriting more than a single instruction.

Table 5.1 16-Bit Re-encoding of Frequent MIPS64 Instructions

Instruction	Major Opcode Name	Number of Register Fields	Immediate Field Size (bit)	Register Field Width (bit)	Total Size of Other Fields	Empty 0 Field Size (bit)	Minor Opcode Size (bit)	Comment
ADDIUS5	POOL16D	5bit:1	4	5		0	1	Add Immediate Unsigned Word Same Register
ADDIUSP	POOL16D	0	9	0		0	1	Add Immediate Unsigned Word to Stack Pointer
ADDIUR2	POOL16E	2	3	3		0	1	Add Immediate Unsigned Word Two Registers

Table 5.1 16-Bit Re-encoding of Frequent MIPS64 Instructions (Continued)

Instruction	Major Opcode Name	Number of Register Fields	Immediate Field Size (bit)	Register Field Width (bit)	Total Size of Other Fields	Empty 0 Field Size (bit)	Minor Opcode Size (bit)	Comment
ADDIUR1SP	POOL16E	1	6	3		0	1	Add Immediate Unsigned Word One Registers and Stack Pointer
ADDU16	POOL16A	3	0	3		0	1	Add Unsigned Word
AND16	POOL16C	2	0	3		0	4	AND
ANDI16	ANDI16	2	4	3		0	0	AND Immediate
B16	B16	0	10			0	0	Branch
BREAK16	POOL16C	0	0		4	0	6	Cause Breakpoint Exception
JALR16	POOL16C	1	0	5		0	5	Jump and Link Register, 32-bit delay-slot
JALRS16	POOL16C	1	0	5		0	5	Jump and Link Register, 16-bit delay-slot
JR16	POOL16C	1	0	5		0	5	Jump Register
LBU16	LBU16	2	4	3		0	0	Load Byte Unsigned
LHU16	LHU16	2	4	3		0	0	Load Halfword
LI16	LI16	1	7	3		0	0	Load Immediate
LW16	LW16	2	4	3		0	0	Load Word
LWGP	LWGP16	1	7	3		0	0	Load Word GP
LWSP	LWSP16	5bit:1	5	5		0	0	Load Word SP
MFHI16	POOL16C	1	0	5		0	5	Move from HI Register
MFLO16	POOL16C	1	0	5		0	5	Move from LO Register
MOVE16	MOVE16	2	0	5		0	0	Move
NOT16	POOL16C	2	0	3		0	4	NOT
OR16	POOL16C	2	0	3		0	4	OR
SB16	SB16	2	4	3		0	0	Store Byte
SDBBP16	POOL16C	0	0		4	0	6	Cause Debug Breakpoint Exception
SH16	SH16	2	4	3		0	0	Store Halfword
SLL16	POOL16B	2	3	3		0	1	Shift Word Left Logical
SRL16	POOL16B	2	3	3		0	1	Shift Word Right Logical
SUBU16	POOL16A	3	0	3		0	1	Sub Unsigned

Table 5.1 16-Bit Re-encoding of Frequent MIPS64 Instructions (Continued)

Instruction	Major Opcode Name	Number of Register Fields	Immediate Field Size (bit)	Register Field Width (bit)	Total Size of Other Fields	Empty 0 Field Size (bit)	Minor Opcode Size (bit)	Comment
SW16	SW16	2	4	3		0	0	Store Word
SWSP	SWSP16	5bit:1	5	5		0	0	Store Word SP
XOR16	POOL16C	2	0	3		0	4	XOR

5.1.2 Frequent MIPS64 Instruction Sequences

These 16-bit instructions are equivalent to frequently-used short sequences of MIPS64 instructions. The instruction-specific register and immediate value selection are shown in [Table 5.3](#).

Table 5.2 16-Bit Re-encoding of Frequent MIPS64 Instruction Sequences

Instruction	Major Opcode Name	Number of Register Fields	Immediate Field Size (bit)	Register Field Width (bit)	Total Size of Other Fields	Empty 0 Field Size (bit)	Minor Opcode Size (bit)	Comment
BEQZ16	BEQZ16	1	7	3		0	0	Branch on Equal Zero
BNEZ16	BNEZ16	1	7	3		0	0	Branch on Not Equal Zero
JRADDIUSP	POOL16C	0	5				5	Jump Register; ADDIU SP
JRC	POOL16C	1	0	5		0	5	Jump Register Compact
LWM16	POOL16C	0	4		2	0	4	Load Word Multiple
MOVEP	POOL16F	3 (encoded)	0	3(encoded)		0	1	Move Register Pair
SWM16	POOL16C	0	4		2	0	4	Store Word Multiple

5.1.3 Instruction-Specific Register Specifiers and Immediate Field Encodings

Table 5.3 Instruction-Specific Register Specifiers and Immediate Field Values

Instruction	Number of Register Fields	Immediate Field Size (bit)	Register 1 Decoded Value	Register 2 Decoded Value	Register 3 Decoded Value	Immediate Field Decoded Value
ADDIUS5	5bit:1	4	rd: 5 bit field			-8..0..7
ADDIUSP	0	9				(-258..-3, 2..257) << 2
ADDIUR2	2	3	rs1:2-7,16, 17	rd:2-7,16, 17		-1, 1, 4, 8, 12, 16, 20, 24
ADDIUR1SP	1	6	rd:2-7,16, 17			(0..63) << 2
ADDU16	3	0	rs1:2-7,16, 17	rs2:2-7,16, 17	rd:2-7,16, 17	
AND16	2	0	rs1:2-7,16, 17	rd:2-7,16, 17		
ANDI16	2	4	rs1:2-7,16, 17	rd:2-7,16, 17		1, 2, 3, 4, 7, 8, 15, 16, 31, 32, 63, 64, 128, 255, 32768, 65535
B16	0	10				(-512..511) << 1
BEQZ16	1	7	rs1:2-7,16, 17			(-64..63) << 1
BNEZ16	1	7	rs1:2-7,16, 17			(-64..63) << 1
BREAK16	0	4				0..15
JALR16	5bit:1	0	rs1:5 bit field			
JALRS16	5bit:1	0	rs1:5 bit field			
JRADDIUSP	0	5				(0..31) << 2
JR16	5bit:1	0	rs1:5 bit field			
JRC	5bit:1	0	rs1:5 bit field			
LBU16	2	4	rb:2-7,16,17	rd:2-7,16, 17		-1,0..14
LHU16	2	4	rb:2-7,16,17	rd:2-7,16, 17		(0..15) << 1
LI16	1	7	rd:2-7,16, 17			-1,0..126
LW16	2	4	rb:2-7,16,17	rd:2-7,16, 17		(0..15) << 2
LWM16	2bit list:1	4				(0..15)<<2
LWGP	1	7	rd:2-7,16,17			(-64..63)<<2
LWSP	5bit:1	5	rd:5-bit field			(0..31)<<2
MFHI16	5bit:1	0	rd:5-bit field			
MFLO16	5bit:1	0	rd:5-bit field			
MOVE16	5bit:2	0	rd:5-bit field	rs1:5-bit field		
MOVEP	3	0	rd, re: (5,6),(5,7),(6,7), (4,21),(4,22),(4, 5),(4,6),(4,7)	rt:0,2,7,16-20	rs:0,2,7,16-20	
NOT16	2	0	rs1:2-7,16, 17	rd:2-7,16, 17		
OR16	2	0	rs1:2-7,16, 17	rd:2-7,16, 17		
SB16	2	4	rb:2-7,16,17	rs1:0, 2-7, 17		0..15

Table 5.3 Instruction-Specific Register Specifiers and Immediate Field Values (Continued)

Instruction	Number of Register Fields	Immediate Field Size (bit)	Register 1 Decoded Value	Register 2 Decoded Value	Register 3 Decoded Value	Immediate Field Decoded Value
SDBBP16	0	0				0..15
SH16	2	4	rb:2-7,16,17	rs1:0, 2-7, 17		(0..15) << 1
SLL16	2	3	rs1:2-7,16, 17	rd:2-7,16, 17		1..8 (see encoding tables)
SRL16	2	3	rs1:2-7,16, 17	rd:2-7,16, 17		1..8 (see encoding tables)
SUBU16	3	0	rs1:2-7,16, 17	rs2:2-7,16, 17	rd:2-7,16, 17	
SW16	2	4	rb:2-7,16,17	rs1:0, 2-7, 17		(0..15) << 2
SWSP	5bit:1	5	rs1: 5 bit field			(0..31) << 2
SWM16	2 bit list:1	4				(0..15)<<2
XOR16	2	0	rs1:2-7,16, 17	rd:2-7,16, 17		

5.2 16-bit Instruction Register Set

Many of the 16-bit instructions use 3-bit register specifiers in their binary encodings. The register set used for most of these 3-bit register specifiers is listed in Table 5.5. The register set used for SB16, SH16, SW16 source register is listed in Table 5.5. These register sets are a true subset of the register set available in 32-bit mode; the 3-bit register specifiers can directly access 8 of the 32 registers available in 32-bit mode (which uses 5-bit register specifiers).

In addition, specific instructions in the 16-bit instruction set implicitly reference the stack pointer register (*sp*), global pointer register (*gp*), the return address register (*ra*), the integer multiplier/divider output registers (*HI/LO*) and the program counter (*PC*). Of these, Table 5.6 lists *sp*, *gp* and *ra*. Table 5.7 lists the microMIPS special-purpose registers, including *PC*, *HI* and *LO*.

The microMIPS also contains some 16-bit instructions that use 5-bit register specifiers. Such 16-bit instructions provide access to all 32 general-purpose registers.

Table 5.4 16-Bit Instruction General-Purpose Registers - \$2-\$7, \$16, \$17

16-Bit Register Encoding ¹	32-Bit MIPS Register Encoding ²	Symbolic Name (From <i>ArchDefs.h</i>)	Description
0	16	s0	General-purpose register
1	17	s1	General-purpose register
2	2	v0	General-purpose register
3	3	v1	General-purpose register
4	4	a0	General-purpose register
5	5	a1	General-purpose register
6	6	a2	General-purpose register

Table 5.4 16-Bit Instruction General-Purpose Registers - \$2-\$7, \$16, \$17 (Continued)

16-Bit Register Encoding ¹	32-Bit MIPS Register Encoding ²	Symbolic Name (From <i>ArchDefs.h</i>)	Description
7	7	a3	General-purpose register

1. “0-7” correspond to the register’s 16-bit binary encoding and show how that encoding relates to the MIPS registers. “0-7” never refer to the registers, except within the binary microMIPS instructions. From the assembler, only the MIPS names (\$16, \$17, \$2, etc.) or the symbolic names (s0, s1, v0, etc.) refer to the registers. For example, to access register number 17 in the register file, the programmer references \$17 or s1, even though the microMIPS binary encoding for this register is 001.
2. General registers not shown in the above table are not accessible through the 16-bit instruction using 3-bit register specifiers. The Move instruction can access all 32 general-purpose registers.

Table 5.5 SB16, SH16, SW16 Source Registers - \$0, \$2-\$7, \$17

16-Bit Register Encoding ¹	32-Bit MIPS Register Encoding ²	Symbolic Name (From <i>ArchDefs.h</i>)	Description
0	0	zero	Hard-wired Zero
1	17	s1	General-purpose register
2	2	v0	General-purpose register
3	3	v1	General-purpose register
4	4	a0	General-purpose register
5	5	a1	General-purpose register
6	6	a2	General-purpose register
7	7	a3	General-purpose register

1. “0-7” correspond to the register’s 16-bit binary encoding and show how that encoding relates to the MIPS registers. “0-7” never refer to the registers, except within the binary microMIPS instructions. From the assembler, only the MIPS names (\$16, \$17, \$2, etc.) or the symbolic names (s0, s1, v0, etc.) refer to the registers. For example, to access register number 17 in the register file, the programmer references \$17 or s1, even though the microMIPS binary encoding for this register is 001.
2. General registers not shown in the above table are not accessible through the 16-bit instructions using 3-bit register specifier. The Move instruction can access all 32 general-purpose registers.

Table 5.6 16-Bit Instruction Implicit General-Purpose Registers

16-Bit Register Encoding	32-Bit MIPS Register Encoding	Symbolic Name (From <i>ArchDefs.h</i>)	Description
Implicit	28	gp	Global pointer register
Implicit	29	sp	Stack pointer register
Implicit	31	ra	Return address register

Table 5.7 16-Bit Instruction Special-Purpose Registers

Symbolic Name	Purpose
PC	Program counter. The PC-relative ADDIU can access this register as an operand.
HI	Contains high-order word of multiply or divide result.
LO	Contains low-order word of multiply or divide result.

5.3 32-Bit Category

5.3.1 New 32-bit instructions

The following table lists the 32-bit instructions introduced in the microMIPS ISA.

Table 5.8 32-bit Instructions introduced within microMIPS

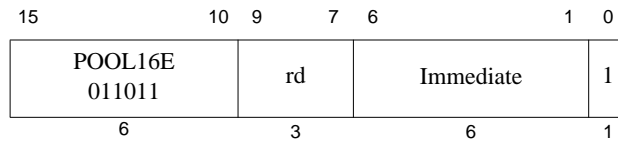
Instruction	Major Opcode Name	Number of Register Fields	Immediate Field Size (bit)	Register Field Width (bit)	Total Size of Other Fields	Empty 0 Field Size (bit)	Minor Opcode Size (bit)	Comment
ADDIUPC	ADDIUPC	1	23	3		0	0	ADDIU PC-Relative
BEQZC	POOL32I	2:5 bit	16	5			0	Branch on Equal to Zero, No Delay Slot
BNEZC	POOL32I	2:5 bit	16	5			0	Branch on Not Equal to Zero, No Delay Slot
JALRS	POOL32A	2:5 bit	0	5			16	Jump and Link Register, Short Delay Slot
JALRS.HB	POOL32A	2:5 bit	0	5			16	Jump and Link Register with Hazard Barrier, Short Delay Slot

Table 5.8 32-bit Instructions introduced within microMIPS (Continued)

Instruction	Major Opcode Name	Number of Register Fields	Immediate Field Size (bit)	Register Field Width (bit)	Total Size of Other Fields	Empty 0 Field Size (bit)	Minor Opcode Size (bit)	Comment
JALS	JALS32	0	26				0	Jump and Link, Short Delay Slot
JALX	JALX		26	5		0	5	Jump and Link Exchange
LDM	POOL32B	1:5bit	12		5	0	4	Load DoubleWord Multiple
LDP	POOL32B	2:5 bit	12			0	4	Load DoubleWord Pair
LWP	POOL32B	2:5 bit	12		5	0	4	Load Word Pair
LWXS	POOL32A	3:5 bit	0	5	0	1	10	Load Word Indexed, Scale
LWM32	POOL32B	1:5bit	12		5	0	4	Load Word Multiple
SDM	POOL32B	1:5bit	12		5	0	4	Store DoubleWord Multiple
SDP	POOL32B	2:5 bit	12			0	4	Load DoubleWord Pair
SWP	POOL32B	2:5 bit	12			0	4	Load Word Pair
SWM32	POOL32B	1:5bits	12		5	0	4	Store Word Multiple

5.4 New Instructions

This section defines all new instructions introduced with microMIPS. Existing MIPS64 instructions and macros are not covered.



Format: ADDIUR1SP rd, decoded_immediate_value

microMIPS

Purpose: Add Immediate Unsigned Word One Register (16-bit instr size)

To add a constant to a 32-bit integer.

Description: $GPR[rd] \leftarrow GPR[29] + \text{zero_extend}(\text{immediate} \ll 2)$

The 6-bit *immediate* field is first shifted left by two bits and then zero-extended. This amount is added to the 32-bit value in GPR 29 and the 32-bit arithmetic result is sign-extended and placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

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[29] + zero_extend(immediate || 02)
GPR[rd] ← sign_extend(temp31..0)

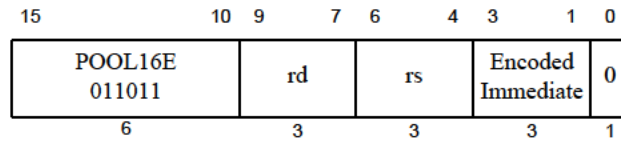
```

Exceptions:

None

Programming Notes:

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



Format: ADDIUR2 rd, rs1, decoded_immediate_value

microMIPS

Purpose: Add Immediate Unsigned Word Two Registers (16-bit instr size)

To add a constant to a 32-bit integer.

Description: $GPR[rd] \leftarrow GPR[rs] + \text{sign_extend}(\text{decoded immediate})$

The encoded immediate field is decoded to obtain the actual immediate value.

The decoded immediate value is sign-extended and then 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.

Table 5.9 Encoded and Decoded Values of the Immediate Field

Encoded Value of Instr _{3..1} (Decimal)	Encoded Value of Instr _{3..1} (Hex)	Decoded Value of Immediate (Decimal)	Decoded Value of Immediate (Hex)
0	0x0	1	0x0001
1	0x1	4	0x0004
2	0x2	8	0x0008
3	0x3	12	0x000c
4	0x4	16	0x0010
5	0x5	20	0x0014
6	0x6	24	0x0018
7	0x7	-1	0xffff

Restrictions:

The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

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(decoded immediate)
GPR[rd] ← sign_extend(temp31..0)

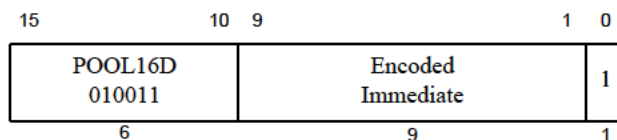
```

Exceptions:

None

Programming Notes:

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



Format: ADDIUSP `decoded_immediate_value`

microMIPS

Purpose: Add Immediate Unsigned Word to Stack Pointer(16-bit instr size)

To add a constant to the stack pointer.

Description: $GPR[29] \leftarrow GPR[29] + \text{sign_extend}(\text{decoded_immediate} \ll 2)$

The encoded immediate field is decoded to obtain the actual immediate value.

The actual immediate value is first shifted left by two bits and then sign-extended. This amount is added to the 32-bit value in GPR 29, and the 32-bit arithmetic result is sign-extended and placed into GPR 29.

No Integer Overflow exception occurs under any circumstances.

Table 5.10 Encoded and Decoded Values of Immediate Field

Encoded Value of Instr _{9..1} (Decimal)	Encoded Value of Instr _{9..1} (Hex)	Decoded Value of Immediate (Decimal)	Decoded Value of Immediate (Hex)
0	0x0	256	0x0100
1	0x1	257	0x0101
2	0x2	2	0x0002
3	0x3	3	0x0003
...
254	0xfe	254	0x00fe
255	0xff	255	0x00ff
256	0x100	-256	0xff00
257	0x101	-255	0xff01
...
508	0x1fc	-4	0xfffc
509	0x1fd	-3	0xfffd
510	0x1fe	-258	0xfefe
511	0x1ff	-257	0xfeff

Restrictions:

If GPR 29 does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

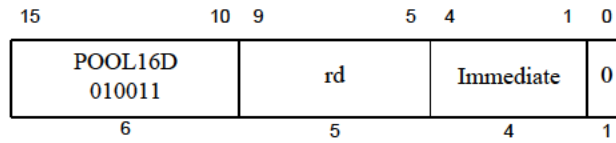
```
if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif
temp ← GPR[29] + sign_extend(decoded immediate || 02)
GPR[29] ← sign_extend(temp31..0)
```

Exceptions:

None

Programming Notes:

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



Format: ADDIUS5 rd, decoded_immediate_value

microMIPS

Purpose: Add Immediate Unsigned Word 5-Bit Register Select (16-bit instr size)

To add a constant to a 32-bit integer

Description: $GPR[rd] \leftarrow GPR[rd] + \text{sign_extend}(\text{immediate})$

The 4-bit *immediate* field is sign-extended and then added to the 32-bit value in GPR *rd*. The 32-bit arithmetic result is sign-extended and placed into GPR *rd*.

The 5-bit register select allows this 16-bit instruction to use any of the 32 GPRs as the destination register.

No Integer Overflow exception occurs under any circumstances.

Table 5-1 Encoded and Decoded Values of Signed Immediate Field

Encoded Value of Instr _{4..1} (Decimal)	Encoded Value of Instr _{4..1} (Hex)	Decoded Value of Immediate (Decimal)	Decoded Value of Immediate (Hex)
0	0x0	0	0x0000
1	0x1	1	0x0001
2	0x2	2	0x0002
3	0x3	3	0x0003
4	0x4	4	0x0004
5	0x5	5	0x0005
6	0x6	6	0x0006
7	0x7	7	0x0007
8	0x8	-8	0xffff8
9	0x9	-7	0xffff9
10	0xa	-6	0xffffa
11	0xb	-5	0xffffb
12	0xc	-4	0xffffc
13	0xd	-3	0xffffd
14	0xe	-2	0xffffe
15	0xf	-1	0xfffff

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:

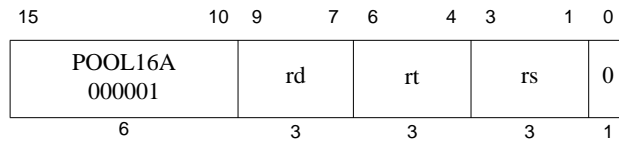
```
if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif
temp ← GPR[rd] + sign_extend(immediate)
GPR[rd] ← sign_extend(temp31..0)
```

Exceptions:

None

Programming Notes:

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



Format: ADDU16 rd, rs, rt

microMIPS

Purpose: Add Unsigned Word (16-bit instr size)

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:

The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

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

Operation:

```

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

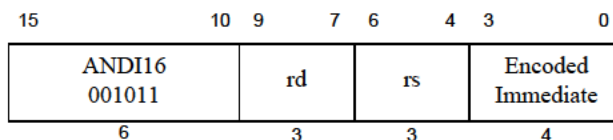
```

Exceptions:

None

Programming Notes:

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



Format: ANDI16 rd, rs, decoded_immediate_value

microMIPS

Purpose: And Immediate (16-bit instr size)

To do a bitwise logical AND with a constant

Description: $GPR[rd] \leftarrow GPR[rs] \text{ AND decoded immediate value}$

The encoded immediate field is decoded to obtain the actual immediate value

The decoded 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 rd.

Table 5-2 Encoded and Decoded Values of Immediate Field

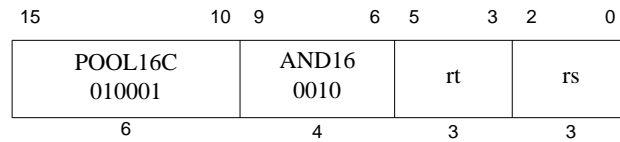
Encoded Value of Instr _{3..0} (Decimal)	Encoded Value of Instr _{3..0} (Hex)	Decoded Value of Immediate (Decimal)	Decoded Value of Immediate (Hex)
0	0x0	128	0x80
1	0x1	1	0x1
2	0x2	2	0x2
3	0x3	3	0x3
4	0x4	4	0x4
5	0x5	7	0x7
6	0x6	8	0x8
7	0x7	15	0xf
8	0x8	16	0x10
9	0x9	31	0x1f
10	0xa	32	0x20
11	0xb	63	0x3f
12	0xc	64	0x40
13	0xd	255	0xff
14	0xe	32768	0x8000
15	0xf	65535	0xffff

Restrictions:

The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

Operation:
$$\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] \text{ and } \text{zero_extend}(\text{decoded immediate})$$
Exceptions:

None



Format: AND16 *rt*, *rs*

microMIPS

Purpose: And (16-bit instr size)

To do a bitwise logical AND

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

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 *rt*.

Restrictions:

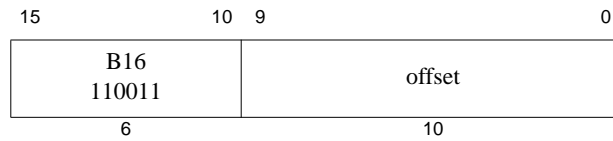
The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

Operation:

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

Exceptions:

None



Format: B16 offset

microMIPS

Purpose: Unconditional Branch (16-bit instr size)

To do an unconditional branch

Description: branch

A 11-bit signed offset (the 10-bit *offset* field shifted left 1 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:

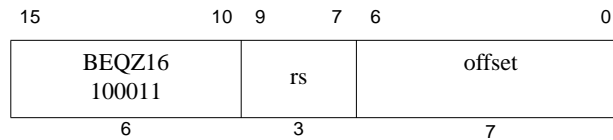
I: $target_offset \leftarrow sign_extend(offset \ll 0^1)$
I+1: $PC \leftarrow PC + target_offset$

Exceptions:

None

Programming Notes:

With the 11-bit signed instruction offset, the branch range is ± 1 Kbytes. Use jump (J) or jump register (JR) or 32-bit branch instructions to branch to addresses outside this range.



Format: BEQZ16 rs, offset

microMIPS

Purpose: Branch on Equal to Zero (16-bit instr size)

To compare a GPR to zero then do a PC-relative conditional branch

Description: if GPR[rs] = 0 then branch

A 8-bit signed offset (the 7-bit *offset* field shifted left 1 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* equals zero, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

The 3-bit register field can only specify GPRs \$2-\$7, \$16, \$17.

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

Operation:

```

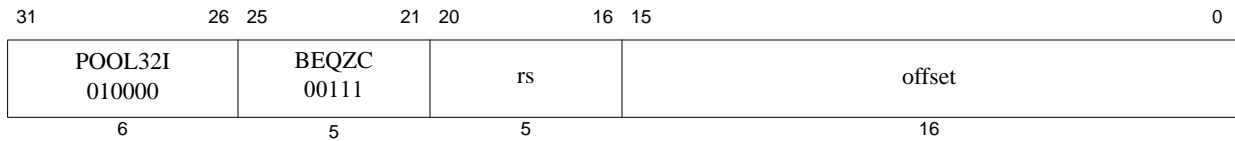
I:    target_offset ← sign_extend(offset || 0)
        condition ← (GPR[rs] == 0)
I+1: if condition then
        PC ← PC + target_offset
        endif
  
```

Exceptions:

None

Programming Notes:

With the 8-bit signed instruction offset, the conditional branch range is ± 64 Bytes. Use 32-bit branch, jump (J) or jump register (JR) instructions to branch to addresses outside this range.



Format: BEQZC *rs*, *offset*

microMIPS

Purpose: Branch on Equal to Zero, Compact

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

Description: if (GPR[*rs*] = 0) then branch

The 16-bit *offset* is shifted left 1 bit, sign-extended, and then added to the address of the instruction after the branch to form the target address. If the contents of GPR *rs* is equal to zero, the program branches to the target address, with no delay slot instruction.

Restrictions:

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

Operation:

```

I:   tgt_offset ← sign_extend(offset || 0)
        condition ← (GPR[rs] = 0GPRLEN)
        if condition then
            PC ← PC + 4 + tgt_offset
        endif

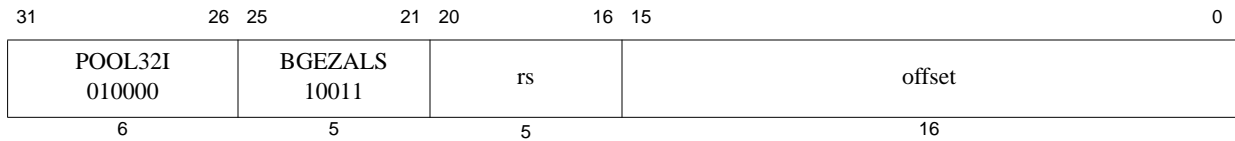
```

Exceptions:

None

Programming Notes:

Unlike most MIPS 'branch' instructions, BEQZC does not have a delay slot.



Format: BGEZALS *rs*, *offset*

microMIPS

Purpose: Branch on Greater Than or Equal to Zero and Link, Short Delay-Slot

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

Description: if $GPR[rs] \geq 0$ 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.

A 17-bit signed offset (the 16-bit *offset* field shifted left 1 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:

The delay-slot instruction must be 16-bits in size. Processor operation is **UNPREDICTABLE** if a 32-bit instruction is placed in the delay slot of BGEZAL.

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 || 01)
        condition ← GPR[rs] ≥ 0GPRLEN
        GPR[31] ← PC + 6
I+1:  if condition then
        PC ← PC + target_offset
        endif

```

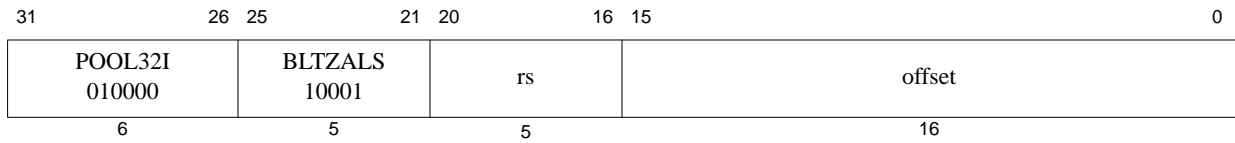
Exceptions:

None

Programming Notes:

With the 17-bit signed instruction offset, the conditional branch range is ± 64 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

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



Format: BLTZALS *rs*, *offset*

microMIPS

Purpose: Branch on Less Than Zero and Link, Short Delay-Slot

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.

A 17-bit signed offset (the 16-bit *offset* field shifted left 1 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:

The delay-slot instruction must be 16-bits in size. Processor operation is **UNPREDICTABLE** if a 32-bit instruction is placed in the delay slot of BLTZAL.

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 || 01)
        condition ← GPR[rs] < 0GPRLEN
        GPR[31] ← PC + 6
I+1: if condition then
        PC ← PC + target_offset
        endif

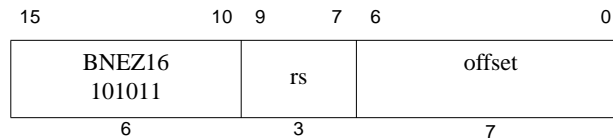
```

Exceptions:

None

Programming Notes:

With the 17-bit signed instruction offset, the conditional branch range is ± 64 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.



Format: BNEZ16 rs, offset

microMIPS

Purpose: Branch on Not Equal to Zero (16-bit instr size)

To compare a GPR to zero then do a PC-relative conditional branch

Description: if GPR[rs] != 0 then branch

A 8-bit signed offset (the 7-bit *offset* field shifted left 1 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* does not equal zero, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

The 3-bit register field can only specify GPRs \$2-\$7, \$16, \$17.

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

Operation:

```

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

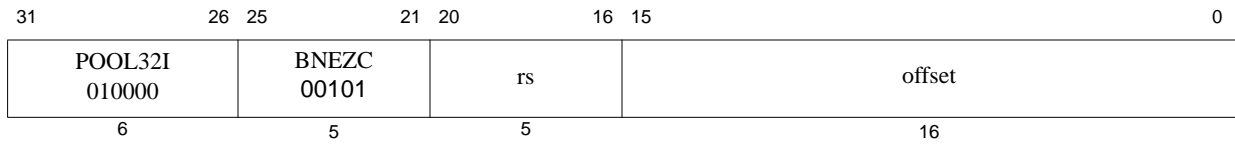
```

Exceptions:

None

Programming Notes:

With the 8-bit signed instruction offset, the conditional branch range is ± 64 Bytes. Use 32-bit branch, jump (J) or jump register (JR) instructions to branch to addresses outside this range.



Format: BNEZC *rs*, *offset*

microMIPS

Purpose: Branch on Not Equal to Zero, Compact

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

Description: if (GPR[*rs*] \neq 0) then branch

The 16-bit *offset* is shifted left 1 bit, sign-extended, and then added to the address of the instruction after the branch to form the target address. If the contents of GPR *rs* is not equal to zero, the program branches to the target address, with no delay slot instruction.

Restrictions:

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

Operation:

```

I:   tgt_offset  $\leftarrow$  sign_extend(offset || 0)
        condition  $\leftarrow$  (GPR[rs]  $\neq$  0GPRLEN)
        if condition then
            PC  $\leftarrow$  PC + 4 + tgt_offset
        endif

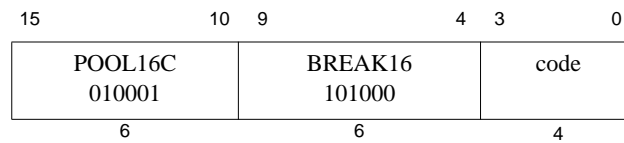
```

Exceptions:

None

Programming Notes:

Unlike most MIPS 'branch' instructions, BNEZC does not have a delay slot.



Format: BREAK16

microMIPS

Purpose: Breakpoint

To cause a Breakpoint exception

Description:

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

Restrictions:

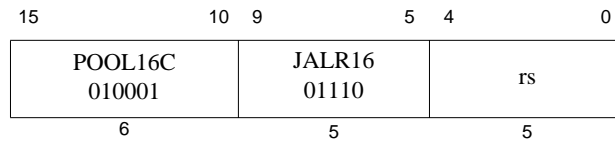
None

Operation:

`SignalException(Breakpoint)`

Exceptions:

Breakpoint



Format: JALR16 rs

microMIPS

Purpose: Jump and Link Register (16-bit instr size)

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

Description: $GPR[31] \leftarrow \text{return_addr}$, $PC \leftarrow GPR[rs]$

Place the return address link in GPR *r31*. 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 MIPS64 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 MIPS64 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.

Restrictions:

The delay-slot instruction must be 32-bits in size. Processor operation is **UNPREDICTABLE** if a 16-bit instruction is placed in the delay slot of JALR16.

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 which implement MIPS64 and if the ISAMode bit of the target is MIPS64 (bit 0 of GPR *rs* is 0) and address bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

For processors that do not implement MIPS64 ISA, if the intended target ISAMode is MIPS64 (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

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

Operation:

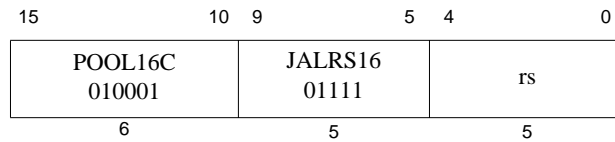
```

I: temp ← GPR[rs]
    GPR[31] ← PC + 6
I+1: if Config3ISA = 1 then
    PC ← temp
    else
    PC ← tempGPRLEN-1..1 || 0
    ISAMode ← temp0
    endif

```

Exceptions:

None



Format: JALRS16 rs

microMIPS

Purpose: Jump and Link Register, Short Delay-Slot(16-bit instr size)

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

Description: $GPR[31] \leftarrow \text{return_addr}$, $PC \leftarrow GPR[rs]$

Place the return address link in GPR *r31*. 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 MIPS64 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 MIPS64 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.

Restrictions:

The delay-slot instruction must be 16-bits in size. Processor operation is **UNPREDICTABLE** if a 32-bit instruction is placed in the delay slot of JALRS16.

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 which implement MIPS64 and if ISAMode bit of the target is MIPS64 (bit 0 of GPR *rs* is 0) and address bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

For processors that do not implement MIPS64 ISA, if the target ISAMode is MIPS64 (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

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

Operation:

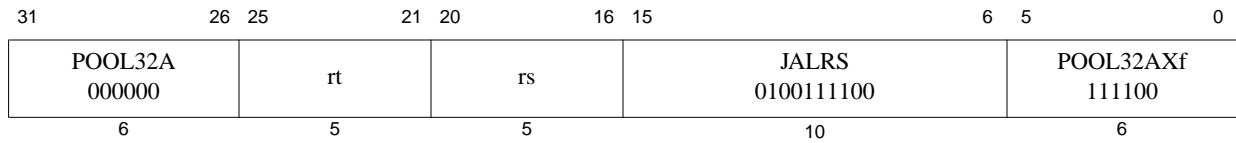
```

I: temp ← GPR[rs]
    GPR[31] ← PC + 4
I+1: if Config3ISA = 1 then
    PC ← temp
    else
    PC ← tempGPRLEN-1..1 || 0
    ISAMode ← temp0
    endif

```


Exceptions:

None



Format: JALRS rs (rt = 31 implied)
JALRS rt, rs

microMIPS
microMIPS

Purpose: Jump and Link Register, Short Delay Slot

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

Description: $GPR[rt] \leftarrow return_addr$, $PC \leftarrow GPR[rs]$

Place the return address link in GPR *rt*. 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 MIPS64 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 MIPS64 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.

Restrictions:

The delay-slot instruction must be 16-bits in size. Processor operation is **UNPREDICTABLE** if a 32-bit instruction is placed in the delay slot of JALRS.

Register specifiers *rs* and *rt* must not be equal, 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.

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 which implement MIPS64 and if if ISAMode bit of the target is MIPS64 (bit 0 of GPR *rs* is 0) and address bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

For processors that do not implement MIPS64 ISA, if the intended target ISAMode is MIPS64 (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

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

Operation:

```

I: temp ← GPR[rs]
    GPR[rt] ← PC + 6
I+1: if Config1CA = 0 then
    PC ← temp
    else

```

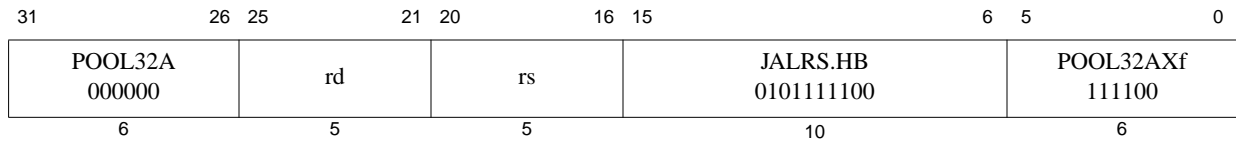
```
    PC ← tempGPRLEN-1..1 || 0  
    ISAMode ← temp0  
endif
```

Exceptions:

None

Programming Notes:

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



Format: JALRS.HB *rs* (*rt* = 31 implied)
JALRS.HB *rt*, *rs*

microMIPS
microMIPS

Purpose: Jump and Link Register with Hazard Barrier, Short Delay-Slot

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

Description: $GPR[rt] \leftarrow return_addr$, $PC \leftarrow GPR[rs]$, clear execution and instruction hazards

Place the return address link in GPR *rt*. 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 MIPS64 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 MIPS64 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.

JALRS.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 JALRS.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 JALRS.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.

Restrictions:

The delay-slot instruction must be 16-bits in size. Processor operation is **UNPREDICTABLE** if a 32-bit instruction is placed in the delay slot of JALRS.HB.

Register specifiers *rs* and *rd* must not be equal, 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.

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 which implement MIPS64 and if ISAMode bit of the target is MIPS64 (bit 0 of GPR *rs* is 0) and address bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

For processors that do not implement MIPS64 ISA, if the intended target ISAMode is MIPS64 (bit 0 of GPR *rs* is

zero), an Address Error exception occurs when the jump target is 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, JALRS.HB, JR.HB, ERET, or DERET. Further, the operation is **UNPREDICTABLE** if the mapping of the current instruction stream is modified.

JALRS.HB does not clear hazards created by any instruction that is executed in the delay slot of the JALRS.HB. Only hazards created by instructions executed before the JALR.HB are cleared by the JALRS.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[rt] ← PC + 6
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 *rt*, if omitted in the assembly language instruction, is GPR 31.

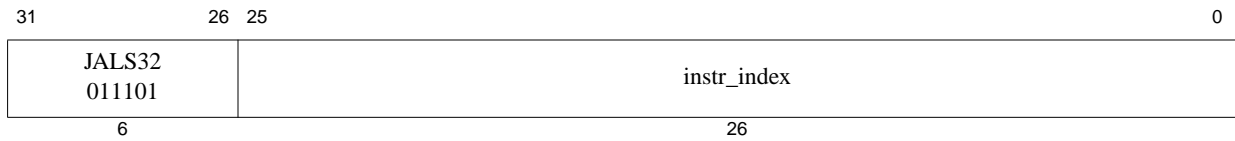
This instruction implements the final step in clearing execution and instruction hazards before execution continues. A hazard is created when a Coprocessor 0 or TLB write affects execution or the mapping of the instruction stream, or after a write to the instruction stream. When such a situation exists, 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, JALRS.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[S][16]) or return (JR) sequence, by simply replacing the original instructions with the HB equivalent.

Example: Clearing hazards due to an ASID change

```

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

```

Format: JALS target

microMIPS

Purpose: Jump and Link, Short Delay Slot

To execute a procedure call within the current 128 MB-aligned region

Description:

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

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 128 MB-aligned region. The low 27 bits of the target address is the *instr_index* field shifted left 1 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

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

Restrictions:

The delay-slot instruction must be 16-bits in size. Processor operation is **UNPREDICTABLE** if a 32-bit instruction is placed in the delay slot of JALS.

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 + 6
I+1: PC ← PC_{GPRELEN-1..27} || instr_index || 0¹

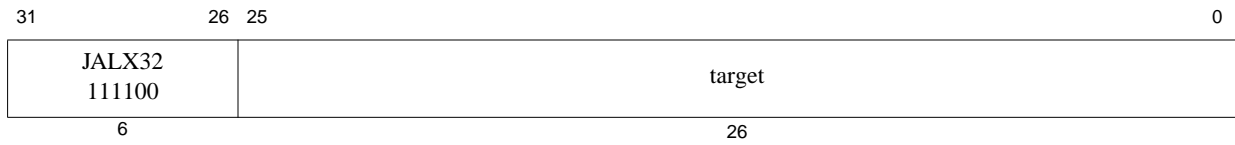
Exceptions:

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 128 MB region aligned on a 128 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 128 MB region, it can branch only to the following 128 MB region containing the branch delay slot.



Format: JALX target

microMIPS

Purpose: Jump and Link Exchange (microMIPS Format)

To execute a procedure call within the current 256 MB-aligned region and change the ISA Mode from microMIPS to 32-bit MIPS.

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 26 bits of the target address is the *target* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction following the branch (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:

The delay-slot instruction must be 32-bits in size. Processor operation is **UNPREDICTABLE** if a 16-bit instruction is placed in the delay slot of JALX.

Processor operation is UNPREDICTABLE if a branch or jump instruction is placed in the delay slot of a jump.

If the MIPS64 ISA is not implemented, a Reserved Instruction Exception is initiated.

Operation:

I: GPR[31] ← (PC + 8)_{GPRLEN-1..1} || ISAMode
I+1: PC ← PC_{GPRLEN-1..28} || target || 0²
 ISAMode ← (not ISAMode)

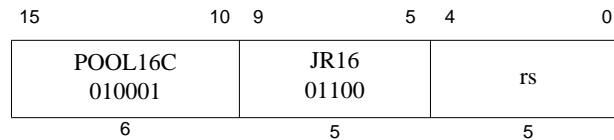
Exceptions:

None

Programming Notes:

Forming the jump target address by concatenating PC and the 26-bit target address 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 jump to anywhere in the region from anywhere in the region which a signed relative *offset* would not allow.

This definition creates the boundary case where the jump instruction is in the last word of a 256 MB region and can therefore jump only to the following 256 MB region containing the following instruction.



Format: JR16 rs

microMIPS

Purpose: Jump Register (16-bit instr size)

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 MIPS64 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 which implement MIPS64 and the ISAMode bit of the target address is MIPS64 (bit 0 of GPR *rs* is 0) and address bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

For processors that do not implement MIPS64 ISA, if the intended target ISAMode is MIPS64 (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

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

Operation:

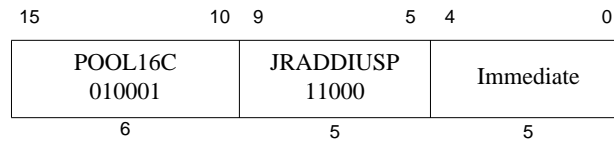
```

I: temp ← GPR[rs]
I+1: if Config3ISA = 1 then
    PC ← temp
    else
    PC ← tempGPRLEN-1..1 || 0
    ISAMode ← temp0
    endif

```

Exceptions:

None



Format: JRADDIUSP decoded_immediate

microMIPS

Purpose: Jump Register, Adjust Stack Pointer (16-bit)

To execute a branch to an instruction address in a register and adjust stack pointer

Description: $PC \leftarrow GPR[ra]; SP \leftarrow SP + \text{zero_extend}(\text{Immediate} \ll 2)$

The program unconditionally jumps to the address specified in GPR 31. If MIPS64 is implemented, the instruction sets the *ISA Mode* bit to the value in GPR 31 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.

The 5-bit *immediate* field is first shifted left by two bits and then zero-extended. This amount is then added to the 32-bit value of GPR 29 and the 32-bit arithmetic result is sign-extended and placed into GPR 29. No Integer Overflow exception occurs under any circumstances for the update of GPR 29.

It is implementation-specific whether interrupts are disabled during the sequence of operations generated by this instruction.

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 which implement MIPS64 and the ISAMode bit of the target address is MIPS64 (bit 0 of GPR *rs* is 0) and address bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

For processors that do not implement MIPS64 ISA, if the intended target ISAMode is MIPS64 (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

If GPR 29 does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```

I:
    PC ← GPR[31]GPREN-1..1 || 0
    if ( Config3ISA > 1 )
        ISAMode ← GPR[31]0
    endif

I+1:
    if NotWordValue(GPR[29]) then
        UNPREDICTABLE
    endif
    temp ← GPR[29] + zero_extend(immediate || 02)
    GPR[29] ← sign_extend(temp31..0)

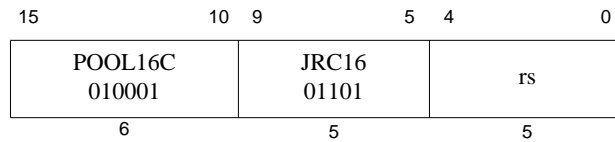
```

Exceptions:

None.

Programming Notes:

Unlike most MIPS “jump” instructions, JRADDIUSP does not have a delay slot.



Format: JRC *rs*

microMIPS

Purpose: Jump Register, Compact (16-bit)

To execute a branch to an instruction address in a register

Description: $PC \leftarrow GPR[rs]$

The program unconditionally jumps to the address specified in GPR *rs*, with no delay slot instruction. If MIPS64 is implemented, the instruction sets the *ISA Mode* bit to the value in GPR *rs* bit 0.

If MIPS64 is implemented, 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 which implement MIPS64 and the ISAMode bit of the target address is MIPS64 (bit 0 of GPR *rs* is 0) and address bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

For processors that do not implement MIPS64 ISA, if the intended target ISAMode is MIPS64 (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

Operation:

```

I: PC ← GPR[rs]GPRELEN-1..1 || 0
      if ( Config3ISA > 1 )
          ISAMode ← GPR[rs]0
      endif

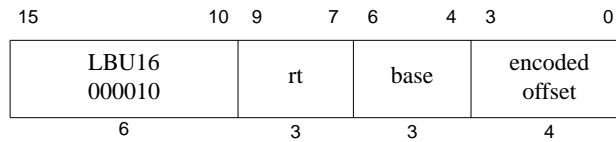
```

Exceptions:

None.

Programming Notes:

Unlike most MIPS “jump” instructions, JRC does not have a delay slot.



Format: LBU16 rt, decoded_offset(base)

microMIPS

Purpose: Load Byte Unsigned (16-bit instr size)

To load a byte from memory as an unsigned value

Description: $GPR[rt] \leftarrow memory[GPR[base] + decoded_offset]$

The encoded offset field is decoded to get the actual offset value. This decoded value is added to the contents of base register to create the effective address. [Table 5.11](#) shows the encoded and decode values of the offset field.

Table 5.11 Offset Field Encoding Range -1, 0..14

Encoded Input (Hex)	Decoded Value (Decimal)
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
a	10
b	11
c	12
d	13
e	14
f	-1

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 4-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

Operation:

$decoded_offset \leftarrow Decode(encoded_offset)$

```
vAddr ← sign_extend(decoded_offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor ReverseEndian3)
memdoubleword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_2..0 xor BigEndianCPU3
GPR[rt] ← zero_extend(memdoubleword7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

for each subsequent register on the list.

It is implementation-specific whether interrupts are disabled during the sequence of operations generated by this instruction.

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.

The behavior of the instruction is **UNDEFINED**, if *base* is included in *reglist*. Reason for this is to allow restartability of the operation if an interrupt or exception has aborted the operation in the middle.

The behavior of this instruction is **UNDEFINED**, if it is placed in a delay slot of a jump or branch.

Operation:

```

if (Are64bitOperationsEnabled() then
  vAddr ← sign_extend(offset) + GPR[base]
  if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
  endif
  for i←0 to fn(reglist)
    (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
    memdoubleword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
    GPR[gpr(reglist,i)] ← memdoubleword
    vAddr ← vAddr + 8
  endfor
else
  SignalException(ReservedInstruction)
endif

function fn(list)
  fn ← number of entries in list - 1
endfunction

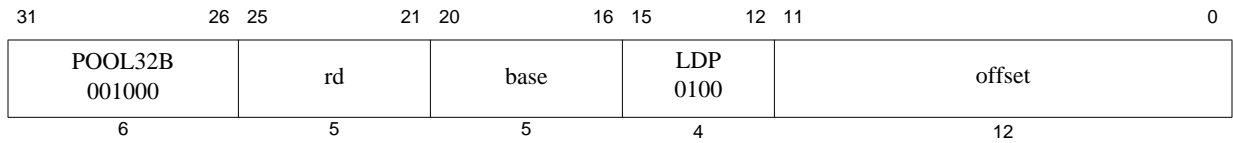
```

Exceptions:

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

Programming Notes:

This instruction executes for a variable number of cycles and performs a variable number of loads from memory. A full restart of the sequence of operations will be performed on return from any exception taken during execution.



Format: LDP rd, offset(base)

microMIPS

Purpose: Load Doubleword Pair

To load two consecutive doublewords from memory

Description: $GPR[rd], GPR[rd+1] \leftarrow \text{memory}[GPR[base] + \text{offset}]$

The contents of the two consecutive 64-bit words at the memory location specified by the aligned effective address are fetched and placed in GPR *rd* and (*rd*+1). The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

It is implementation-specific whether interrupts are disabled during the sequence of operations generated by this instruction.

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.

The behavior of the instructions is **UNDEFINED** if *rd* equals \$31.

The behavior of the instruction is **UNDEFINED**, if *base* and *rd* are the same. Reason for this is to allow restartability of the operation if an interrupt or exception has aborted the operation in the middle.

The behavior of this instruction is **UNDEFINED**, if it is placed in a delay slot of a jump or branch.

Operation:

```

if (Are64bitOperationsEnabled() then
  vAddr ← sign_extend(offset) + GPR[base]
  if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
  endif
  (pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)←
  memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)←
  GPR[rd] ← memdoubleword
  vAddr ← sign_extend(offset) + GPR[base] + 8
  (pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)←
  memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)←
  GPR[rd+1]← memdoubleword
else
  SignalException(ReservedInstruction)
endif

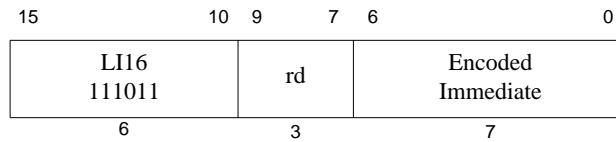
```

Exceptions:

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

Programming Notes:

This instruction may execute for a variable number of cycles and performs two loads from memory. A full restart of the sequence of operations will be performed on return from any exception taken during execution.



Format: LI16 rd, decoded_immediate

microMIPS

Purpose: Load Immediate Word (16-bit instr size)

To load a 6-bit constant into a register.

Description: $GPR[rd] \leftarrow decoded_immediate$

The 7-bit encoded Immediate field is decoded to obtain the actual immediate value. Table 5.12 shows the encoded values of the Immediate field and the actual immediate values.

Table 5.12 LI16 -1, 0..126 Immediate Field Encoding Range

Encoded Input (Hex)	Decoded Value (Decimal)
0	0
1	1
2	2
3	3
...	...
7e	126
7f	-1

The actual decoded immediate value is sign-extended and placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

Operation:

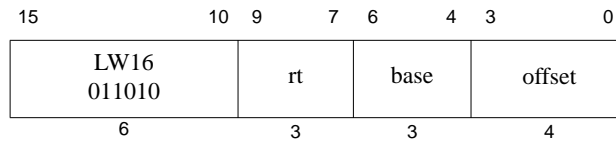
```

decoded_immediate ← Decode(encoded_immediate)
temp ← sign_extend(decoded_immediate)
GPR[rd] ← temp63..0

```

Exceptions:

None



Format: LW16 rt, left_shifted_offset(base)

microMIPS

Purpose: Load Word (16-bit instr size)

To load a word from memory as a signed value

Description: $GPR[rt] \leftarrow \text{memory}[GPR[base] + (\text{offset} \times 4)]$

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 4-bit signed *offset* is left shifted by two bits and then is added to the contents of GPR *base* to form the effective address.

Restrictions:

The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

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:

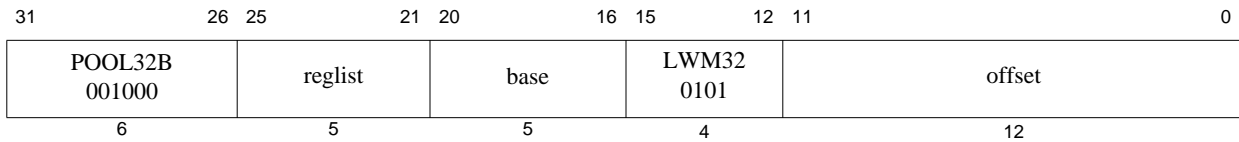
```

vAddr ← sign_extend(offset || 02) + 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)

```

Exceptions:

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



Format: LWM32 {srel6, } {ra}, offset(base)

microMIPS

Purpose: Load Word Multiple

To load a sequence of consecutive words from memory

Description: {GPR[16], {GPR[17], {GPR[18], {GPR[19], {GPR[20], {GPR[21], {GPR[22], {GPR[23], {GPR[30]}}}}}}}} {GPR[31]} ←
memory[GPR[base]+offset], ..., memory[GPR[base]+offset+4*(fn(reglist))]

The contents of consecutive 32-bit words at the memory location specified by the 32-bit aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in the GPRs defined by *reglist*. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The following table shows the encoding of the *reglist* field.

<i>reglist</i> Encoding (binary)	List of Registers Loaded
0 0 0 0 1	GPR[16]
0 0 0 1 0	GPR[16], GPR[17]
0 0 0 1 1	GPR[16], GPR[17], GPR[18]
0 0 1 0 0	GPR[16], GPR[17], GPR[18], GPR[19]
0 0 1 0 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20]
0 0 1 1 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21]
0 0 1 1 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22]
0 1 0 0 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[23]
0 1 0 0 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[23], GPR[30]
1 0 0 0 0	GPR[31]
1 0 0 0 1	GPR[16], GPR[31]
1 0 0 1 0	GPR[16], GPR[17], GPR[31]
1 0 0 1 1	GPR[16], GPR[17], GPR[18], GPR[31]
1 0 1 0 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[31]
1 0 1 0 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[31]
1 0 1 1 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[31]
1 0 1 1 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[31]
1 1 0 0 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[23], GPR[31]
1 1 0 0 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[23], GPR[30], GPR[31]
All other combinations	Reserved

The register numbers and the effective addresses are correlated using the order listed in the table, starting with the

left-most register on the list and ending with the right-most register on the list. The effective address is incremented for each subsequent register on the list.

It is implementation-specific whether interrupts are disabled during the sequence of operations generated by this instruction.

Restrictions:

The effective address must be 32-bit aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

The behavior of the instruction is **UNPREDICTABLE**, if *base* is included in *reglist*. Reason for this is to allow restartability of the operation if an interrupt or exception has aborted the operation in the middle.

The behavior of this instruction is **UNPREDICTABLE**, if it is placed in a delay slot of a jump or branch.

Operation:

```

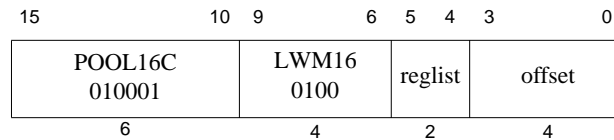
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
for i←0 to fn(reglist)
    (pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
    pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
    memdoubleword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
    byte ← vAddr2..0 xor (BigEndianCPU || 02)
    GPR[gpr(reglist,i)] ← sign_extend(memdoubleword31+8*byte..8*byte)
    vAddr ← vAddr + 4
endfor

function fn(list)
    fn ← (number of entries in list) - 1
endfunction

```

Exceptions:

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



Format: LWM16 s0, {s1, {s2, {s3,}} } ra, left_shifted_offset(sp)

microMIPS

Purpose: Load Word Multiple (16-bit)

To load a sequence of consecutive words from memory

Description: GPR[16], {GPR[17], {GPR[18], {GPR[19],}} } GPR[31] ← memory[GPR[29] + (offset << 2)], ..., memory[GPR[19] + (offset << 2) + 4 * (fn(reglist))]

The contents of consecutive 32-bit words at the memory location specified by the 32-bit aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in the GPRs defined by *reglist*. The 4-bit unsigned *offset* is first left shifted by two bits and then added to the contents of GPR *sp* to form the effective address.

The following table shows the encoding of the *reglist* field.

<i>reglist</i> Encoding (binary)	List of Registers Loaded
0 0	GPR[16], GPR[31]
0 1	GPR[16], GPR[17], GPR[31]
1 0	GPR[16], GPR[17], GPR[18], GPR[31]
1 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[31]

The register numbers and the effective addresses are correlated using the order listed in the table, starting with the left-most register on the list and ending with the right-most register on the list. The effective address is incremented for each subsequent register on the list.

It is implementation-specific whether interrupts are disabled during the sequence of operations generated by this instruction.

Restrictions:

The effective address must be 32-bit aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

The behavior of this instruction is **UNPREDICTABLE**, if it is placed in a delay slot of a jump or branch.

Operation:

```

vAddr ← zero_extend(offset || 02) + GPR[sp]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
for i ← 0 to fn(reglist)
    (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[gpr(reglist, i)] ← sign_extend(memdoubleword31+8*byte..8*byte)
    vAddr ← vAddr + 4

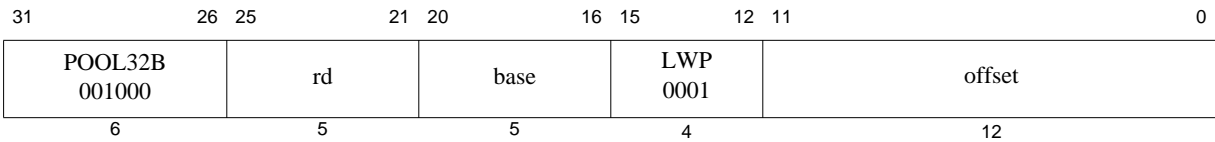
```



```
endfor  
  
function fn(list)  
    fn ← number of entries in list - 1  
endfunction
```

Exceptions:

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



Format: LWP rd, offset(base)

microMIPS

Purpose: Load Word Pair

To load two consecutive words from memory

Description: $GPR[rd], GPR[rd+1] \leftarrow \text{memory}[GPR[base] + \text{offset}]$

The contents of the two consecutive 32-bit words at the memory location specified by the 32-bit aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rd* and (*rd+1*). The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

It is implementation-specific whether interrupts are disabled during the sequence of operations generated by this instruction.

Restrictions:

The effective address must be 32-bit aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

The behavior of the instructions is **UNPREDICTABLE** if *rd* equals r31.

The behavior of the instruction is **UNPREDICTABLE**, if *base* and *rd* are the same. Reason for this is to allow restartability of the operation if an interrupt or exception has aborted the operation in the middle.

The behavior of this instruction is **UNPREDICTABLE**, if it is placed in a delay slot of a jump or branch.

Operation:

```

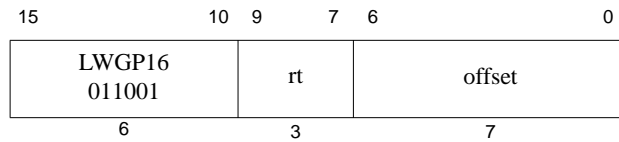
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte ← vAddr2..0 xor (BigEndianCPU || 02)
GPR[rd] ← sign_extend(memdoubleword31+8*byte..8*byte)

vAddr ← sign_extend(offset) + GPR[base] + 4
(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[rd+1] ← sign_extend(memdoubleword31+8*byte..8*byte)

```

Exceptions:

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



Format: LWGP *rt*, left_shifted_offset(*gp*)

microMIPS

Purpose: Load Word from Global Pointer (16-bit instr size)

To load a word from memory as a signed value

Description: $GPR[rt] \leftarrow \text{memory}[GPR[28] + (\text{offset} \times 4)]$

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 7-bit signed *offset* is left shifted by two bits and then added to the contents of GPR 28 to form the effective address.

Restrictions:

The 3-bit register field can only specify GPRs \$2-\$7, \$16, \$17.

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:

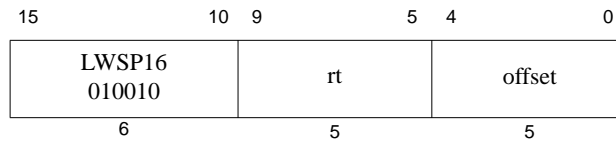
```

vAddr ← sign_extend(offset || 02) + GPR[28]
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)

```

Exceptions:

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



Format: LWSP *rt*, left_shifted_offset(*sp*)

microMIPS

Purpose: Load Word from Stack Pointer (16-bit instr size)

To load a word from memory as a signed value

Description: $GPR[rt] \leftarrow \text{memory}[GPR[29] + (\text{offset} \times 4)]$

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 5-bit signed *offset* is left shifted by two bits, zero-extended and then is added to the contents of GPR 29 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:

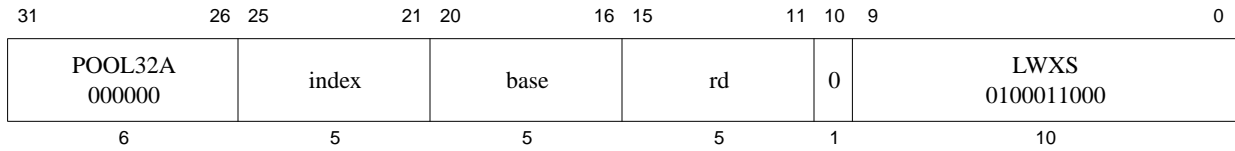
```

vAddr ← zero_extend(offset || 02) + GPR[29]
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)

```

Exceptions:

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



Format: LWXS rd, index(base)

microMIPS

Purpose: Load Word Indexed, Scaled

To load a word from memory as a signed value, using scaled indexed addressing.

Description: $GPR[rd] \leftarrow \text{memory}[GPR[base] + (GPR[index] \times 4)]$

The contents of GPR *index* is multiplied by 4 and the result is added to the contents of GPR *base* to form an effective address. The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rd*.

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:

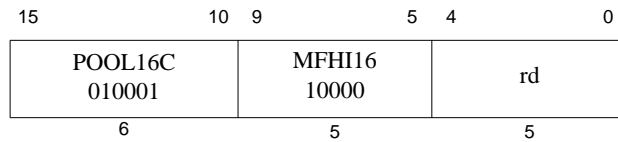
```

vAddr ← (GPR[index]61..0 || 02) + 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))
memdouble ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte ← vAddr2..0 xor (BigEndianCPU || 02)
GPR[rd] ← sign_extend(memdouble31+8*byte..8*byte)

```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error



Format: MFHI16 rd

microMIPS

Purpose: Move From HI Register (16-bit instr size)

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

Description: $GPR[rd] \leftarrow HI$

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

Restrictions:

None

Operation:

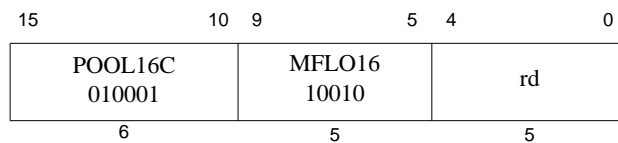
$GPR[rd] \leftarrow HI$

Exceptions:

None

Historical Information:

In the MIPS I, II, and III architectures, the two instructions which follow the MFHI must not modify the *HI* register. If this restriction is violated, the result of the MFHI is **UNPREDICTABLE**. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.



Format: MFLO16 rd

microMIPS

Purpose: Move From LO Register

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

Description: $GPR[rd] \leftarrow LO$

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

Restrictions:

None

Operation:

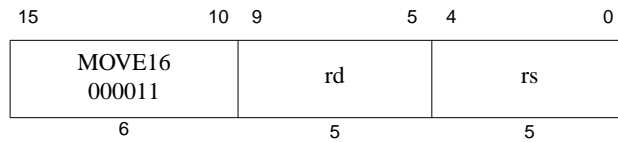
$GPR[rd] \leftarrow LO$

Exceptions:

None

Historical Information:

In the MIPS I, II, and III architectures, the two instructions which follow the MFHI must not modify the *HI* register. If this restriction is violated, the result of the MFHI is **UNPREDICTABLE**. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.



Format: MOVE16 rd, rs

microMIPS

Purpose: Move Register (16-bit instr size)

To copy one GPR to another GPR.

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

The contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

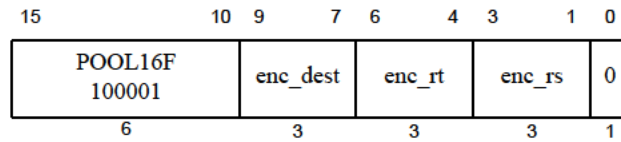
None

Operation:

$GPR[rd] \leftarrow GPR[rs]$

Exceptions:

None



Format: MOVEP *rd*, *re*, *rs*, *rt*

microMIPS

Purpose: Move a Pair of Registers

To copy two GPRs to another two GPRs.

Description: $GPR[rd] \leftarrow GPR[rs]$; $GPR[re] \leftarrow GPR[rt]$;

The contents of GPR *rs* are placed into GPR *rd*. The contents of GPR *rt* are placed into GPR *re*.

The register numbers *rd* and *re* are determined by the encoded *enc_dest* field:

Table 5.13 Encoded and Decoded Values of the Enc_Dest Field

Encoded Value of Instr _{9..7} (Decimal)	Encoded Value of Instr _{9..7} (Hex)	Decoded Value of <i>rd</i> (Decimal)	Decoded Value of <i>re</i> (Decimal)
0	0x0	5	6
1	0x1	5	7
2	0x2	6	7
3	0x3	4	21
4	0x4	4	22
5	0x5	4	5
6	0x6	4	6
7	0x7	4	7

The register numbers *rs* and *rt* are determined by the encoded *enc_rs* and *enc_rt* fields:

Table 5.14 Encoded and Decoded Values of the Enc_rs and Enc_rt Fields

Encoded Value of Instr _{6..4} (or Instr _{3..1}) (Decimal)	Encoded Value of Instr _{6..4} (or Instr _{3..1}) (Hex)	Decoded Value of <i>rt</i> (or <i>rs</i>) (Decimal)	Symbolic Name (From ArchDefs.h)
0	0x0	0	zero
1	0x1	17	s1
2	0x2	2	v0
3	0x3	3	v1
4	0x4	16	s0
5	0x5	18	s2
6	0x6	19	s3

Table 5.14 Encoded and Decoded Values of the Enc_rs and Enc_rt Fields

Encoded Value of Instr _{6..4} (or Instr _{3..1}) (Decimal)	Encoded Value of Instr _{6..4} (or Instr _{3..1}) (Hex)	Decoded Value of rt (or rs) (Decimal)	Symbolic Name (From ArchDefs.h)
7	0x7	20	s4

It is implementation-specific whether interrupts are disabled during the sequence of operations generated by this instruction.

Restrictions:

The destination register pair field, *enc_dest*, can only specify the register pairs defined in Table 5.13.

The source register fields *enc_rs* and *enc_rt* can only specify GPRs 0,2-3,16-20.

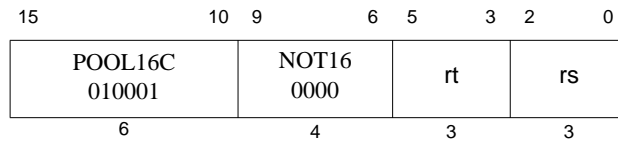
The behavior of this instruction is **UNPREDICTABLE**, if it is placed in a delay slot of a jump or branch.

Operation:

$GPR[rd] \leftarrow GPR[rs]; GPR[re] \leftarrow GPR[rt]$

Exceptions:

None



Format: NOT16 rt, rs

microMIPS

Purpose: Invert (16-bit instr size)

To do a bitwise logical inversion.

Description: $GPR[rt] \leftarrow GPR[rs] \text{ XOR } 0xffffffff$

Invert the contents of GPR *rs* in a bitwise fashion and place the result into GPR *rt*.

Restrictions:

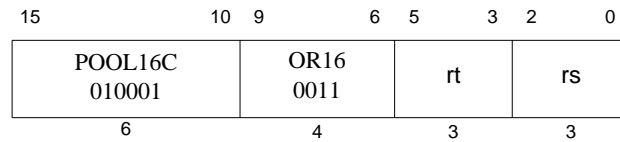
The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

Operation:

$GPR[rt] \leftarrow GPR[rs] \text{ xor } 0xffffffff$

Exceptions:

None



Format: OR16 *rt*, *rs*

MIPS32

Purpose: Or (16-bit instr size)

To do a bitwise logical OR

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

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 *rt*.

Restrictions:

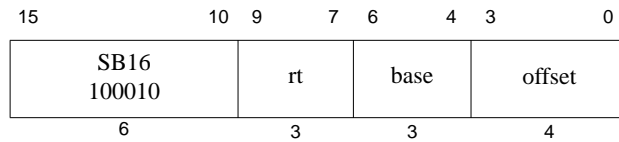
The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

Operation:

$GPR[rt] \leftarrow GPR[rs] \text{ or } GPR[rt]$

Exceptions:

None



Format: SB16 rt, offset(base)

microMIPS

Purpose: Store Byte (16-bit instr size)

To store a byte to memory

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

The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 4-bit unsigned *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The 3-bit *base* register field can only specify GPRs \$2-\$7, \$16, \$17.

The 3-bit *rt* register field can only specify GPRs \$0, \$2-\$7, \$17.

Operation:

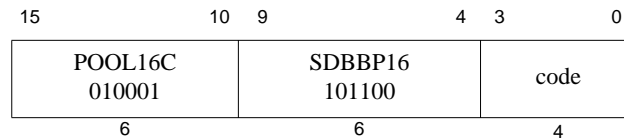
```

vAddr ← zero_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor ReverseEndian3)
bytesel ← vAddr_2..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



Format: SDBBP16 code

EJTAG+microMIPS

Purpose: Software Debug Breakpoint (16-bit instr size)

To cause a debug breakpoint exception

Description:

This instruction causes a debug exception, passing control to the debug exception handler. If the processor is executing in Debug Mode when the SDBBP instruction is executed, the exception is a Debug Mode Exception, which sets the `Debug_DExcCode` field to the value 0x9 (Bp). The code field can be used for passing information to the debug exception handler, and is retrieved by the debug exception handler only by loading the contents of the memory word containing the instruction, using the DEPC register. The CODE field is not used in any way by the hardware.

Restrictions:

Operation:

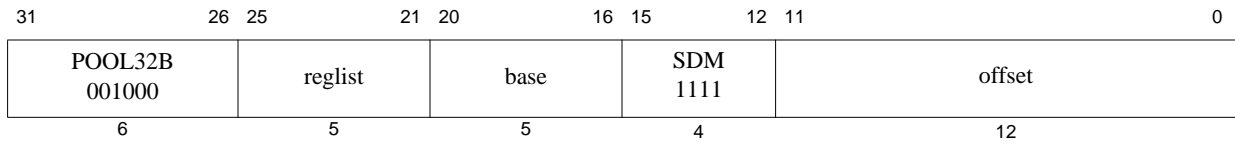
```

If DebugDM = 0 then
    SignalDebugBreakpointException()
else
    SignalDebugModeBreakpointException()
endif

```

Exceptions:

Debug Breakpoint Exception
 Debug Mode Breakpoint Exception



Format: SDM {sregs, } {ra}, offset(base)

microMIPS

Purpose: Store Doubleword Multiple

To store a sequence of consecutive doublewords to memory

Description: $memory[GPR[base]+offset], \dots, memory[GPR[base]+offset+4*(fn(reglist))]$ ← {GPR[16], {GPR[17], {GPR[18], {GPR[19], {GPR[20], {GPR[21], {GPR[22], {GPR[23], {GPR[30]}}}}}}}} {GPR[21]}

The contents of the 64-bit doublewords of the GPRs defined by *reglist* are stored in memory at the location specified by the aligned effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The following table shows the encoding of the *reglist* field.

reglist Encoding (binary)	List of Registers Stored
0 0 0 0 1	GPR[16]
0 0 0 1 0	GPR[16], GPR[17]
0 0 0 1 1	GPR[16], GPR[17], GPR[18]
0 0 1 0 0	GPR[16], GPR[17], GPR[18], GPR[19]
0 0 1 0 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20]
0 0 1 1 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21]
0 0 1 1 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22]
0 1 0 0 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[23]
0 1 0 0 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[23], GPR[30]
1 0 0 0 0	GPR[31]
1 0 0 0 1	GPR[16], GPR[31]
1 0 0 1 0	GPR[16], GPR[17], GPR[31]
1 0 0 1 1	GPR[16], GPR[17], GPR[18], GPR[31]
1 0 1 0 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[31]
1 0 1 0 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[31]
1 0 1 1 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[31]
1 0 1 1 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[31]
1 1 0 0 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[23], GPR[31]
1 1 0 0 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[23], GPR[30], GPR[31]
All other combinations	Reserved

The register numbers and the effective addresses are correlated using the order listed in the table, starting with the left-most register on the list and ending with the right-most register on the list. The effective address is incremented

for each subsequent register on the list.

It is implementation-specific whether interrupts are disabled during the sequence of operations generated by this instruction.

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.

The behavior of this instruction is **UNDEFINED**, if it is placed in a delay slot of a jump or branch.

Operation:

```

if Are64bitOperationsEnabled() then
  vAddr ← sign_extend(offset) + GPR[base]
  if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
  endif
  for i←0 to fn(reglist)
    (pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
    datadoubleword ← GPR[gpr(reglist,i)]
    StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
    vAddr ← vAddr + 8
  endfor
else
  SignalException(ReservedInstruction)
endif

function fn(list)
  fn ← number of entries in list - 1
endfunction

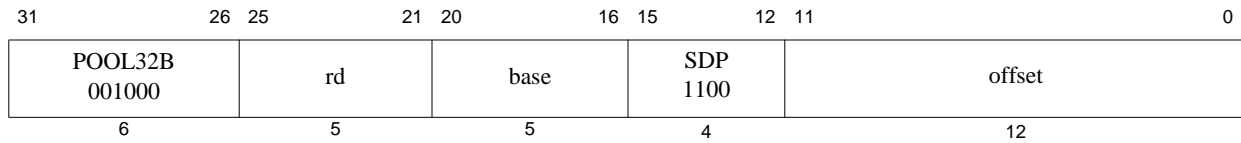
```

Exceptions:

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

Programming Notes:

This instruction executes for a variable number of cycles and performs a variable number of stores to memory. A full restart of the sequence of operations will be performed on return from any exception taken during execution.



Format: SDP rd, offset(base)

microMIPS

Purpose: Store Doubleword Pair

To store two consecutive doublewords to memory

Description: $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{GPR}[\text{rd}], \text{GPR}[\text{rd}+1]$

The contents of the 64-bit doublewords of GPR *rd* and GPR *rd+1* are stored in memory at the location specified by the aligned effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

It is implementation-specific whether interrupts are disabled during the sequence of operations generated by this instruction.

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.

The behavior of the instructions is **UNDEFINED** if *rd* equals \$31.

The behavior of this instruction is **UNDEFINED**, if it is placed in a delay slot of a jump or branch.

Operation:

```

if Are64bitOperationsEnabled() then
  vAddr ← sign_extend(offset) + GPR[base]
  if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
  endif
  (pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)←←
  datadoubleword ← GPR[rd]
  StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
  vAddr ← sign_extend(offset) + GPR[base] + 8
  (pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)←←
  datadoubleword ← GPR[rd+1]
  StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
else
  SignalException(ReservedInstruction)
endif

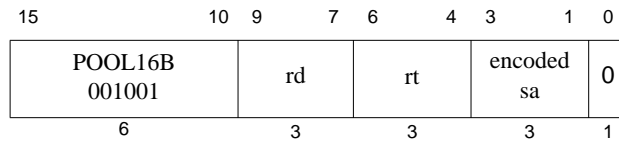
```

Exceptions:

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

Programming Notes:

This instruction may execute for a variable number of cycles and may perform a variable number of stores to memory. A full restart of the sequence of operations will be performed on return from any exception taken during execution.



Format: SLL16 rd, rt, decoded_sa

microMIPS

Purpose: Shift Word Left Logical (16-bit instr size)

To left-shift a word by a fixed number of bits

Description: $GPR[rd] \leftarrow GPR[rt] \ll decoded_sa$

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 decoding the *encoded_sa* field. Table 5.15 lists the encoded values of the *encoded_sa* field and the actual bit shift amount values.

Table 5.15 Shift Amount Field Encoding

Encoded Input (Hex)	Decoded Value (Decimal)
0	8
1	1
2	2
3	3
4	4
5	5
6	6
7	7

Restrictions:

The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

Operation:

```

decoded_sa ← DECODE(encoded_sa)
s ← decoded_sa
temp ← GPR[rt](31-s)..0 || 0s
GPR[rd] ← sign_extend(temp)

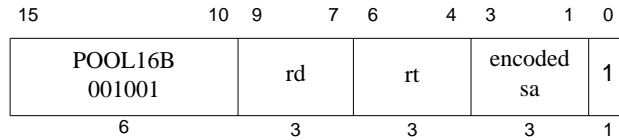
```

Exceptions:

None

Programming Notes:

Unlike nearly all other word operations, the SLL input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign-extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign-extends it.



Format: SRL16 rd, rt, decoded_sa

microMIPS

Purpose: Shift Word Right Logical (16-bit instr size)

To execute a logical right-shift of a word by a fixed number of bits

Description: $GPR[rd] \leftarrow GPR[rt] \gg decoded_sa$ (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 *s* by decoding the *encoded_sa* field. Table 5.16 lists the encoded values of the *encoded_sa* field and the actual bit shift amount values.

Table 5.16 Shift Amount Field Encoding

Encoded Input (Hex)	Decoded Value (Decimal)
0	8
1	1
2	2
3	3
4	4
5	5
6	6
7	7

Restrictions:

The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

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:

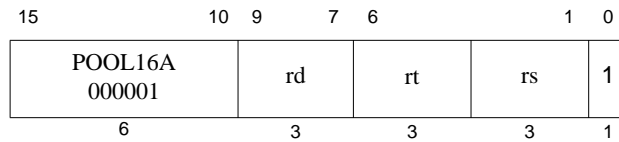
```

if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
decoded_sa ← DECODE(encoded_sa)
s ← decoded_sa
temp ← 0s || GPR[rt]31..s
GPR[rd] ← sign_extend(temp)

```

Exceptions:

None



Format: SUBU16 rd, rs, rt

microMIPS

Purpose: Subtract Unsigned Word (16-bit instr size)

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:

The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

On 64-bit processors, if either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```

if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← GPR[rs] - GPR[rt]
GPR[rd] ← sign_extend(temp)

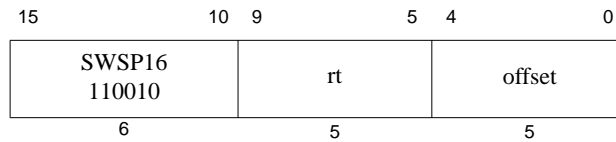
```

Exceptions:

None

Programming Notes:

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



Format: SWSP *rt*, left_shifted_offset(*base*)

microMIPS

Purpose: Store Word to Stack Pointer (16-bit instr size)

To store a word to memory

Description: $\text{memory}[\text{GPR}[29] + (\text{offset} \times 4)] \leftarrow \text{GPR}[\text{rt}]$

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 5-bit signed *offset* is left shifted by two bits, zero-extended and then is added to the contents of GPR 29 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:

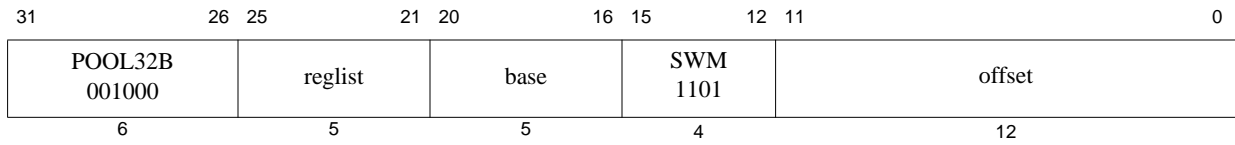
```

vAddr ← zero_extend(offset || 02) + GPR[29]
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
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch



Format: SWM32 {sregs, } {ra}, offset(base)

microMIPS

Purpose: Store Word Multiple

To store a sequence of consecutive words to memory

Description: $memory[GPR[base]+offset], \dots, memory[GPR[base]+offset+4*(fn(reglist))] \leftarrow \{GPR[16], \{GPR[17], \{GPR[18], \{GPR[19], \{GPR[20], \{GPR[21], \{GPR[22], \{GPR[23], \{GPR[30]\}\}\}\}\}\}\}\}\{GPR[31]\}$

The least-significant 32-bit words of the GPRs defined by *reglist* are stored in memory at the location specified by the aligned effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The following table shows the encoding of the *reglist* field.

<i>reglist</i> Encoding (binary)	List of Registers Loaded
0 0 0 0 1	GPR[16]
0 0 0 1 0	GPR[16], GPR[17]
0 0 0 1 1	GPR[16], GPR[17], GPR[18]
0 0 1 0 0	GPR[16], GPR[17], GPR[18], GPR[19]
0 0 1 0 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20]
0 0 1 1 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21]
0 0 1 1 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22]
0 1 0 0 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[23]
0 1 0 0 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[23], GPR[30]
1 0 0 0 0	GPR[31]
1 0 0 0 1	GPR[16], GPR[31]
1 0 0 1 0	GPR[16], GPR[17], GPR[31]
1 0 0 1 1	GPR[16], GPR[17], GPR[18], GPR[31]
1 0 1 0 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[31]
1 0 1 0 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[31]
1 0 1 1 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[31]
1 0 1 1 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[31]
1 1 0 0 0	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[23], GPR[31]
1 1 0 0 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[20], GPR[21], GPR[22], GPR[23], GPR[30], GPR[31]
All other combinations	Reserved

The register numbers and the effective addresses are correlated using the order listed in the table, starting with the left-most register on the list and ending with the right-most register on the list. The effective address is incremented

for each subsequent register on the list.

It is implementation-specific whether interrupts are disabled during the sequence of operations generated by this instruction.

Restrictions:

The effective address must be 32-bit aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

The behavior of this instruction is **UNPREDICTABLE**, if it is placed in a delay slot of a jump or branch.

Operation:

```

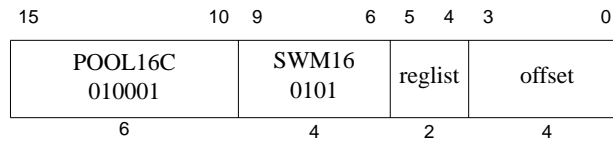
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
for i←0 to fn(reglist)
    (pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
    pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
    bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
    datadoubleword ← GPR[gpr(reglist,i)]63-8*bytesel..0 || 08*bytesel
    StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
    vAddr ← vAddr + 4
endfor

function fn(list)
    fn ← (number of entries in list) - 1
endfunction

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch



Format: SWM16 s0, {s1, {s2, {s3,}}} ra, left_shifted_offset(sp)

microMIPS

Purpose: Store Word Multiple (16-bit)

To store a sequence of consecutive words to memory

Description: $\text{memory}[\text{GPR}[29]], \dots, \text{memory}[\text{GPR}[29] + (\text{offset} \ll 2) + 4 * (2 + \text{fn}(\text{reglist}))] \leftarrow \text{GPR}[16], \{\text{GPR}[17], \{\text{GPR}[18], \{\text{GPR}[19], \dots\}\}\} \text{GPR}[31]$

The least-significant 32-bit words of the GPRs defined by *reglist* are stored in memory at the location specified by the aligned effective address. The 4-bit unsigned *offset* is added to the contents of GPR *sp* to form the effective address.

The following table shows the encoding of the *reglist* field.

<i>reglist</i> Encoding (binary)	List of Registers Stored
0 0	GPR[16], GPR[31]
0 1	GPR[16], GPR[17], GPR[31]
1 0	GPR[16], GPR[17], GPR[18], GPR[31]
1 1	GPR[16], GPR[17], GPR[18], GPR[19], GPR[31]

The register numbers and the effective addresses are correlated using the order listed in the table, starting with the left-most register on the list and ending with the right-most register on the list. The effective address is incremented for each subsequent register on the list.

It is implementation-specific whether interrupts are disabled during the sequence of operations generated by this instruction.

Restrictions:

The effective address must be 32-bit aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

The behavior of this instruction is **UNPREDICTABLE**, if it is placed in a delay slot of a jump or branch.

Operation:

```

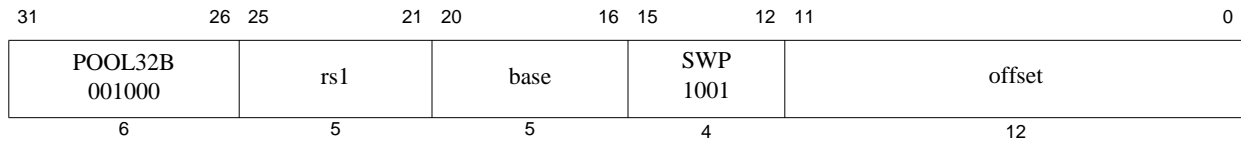
vAddr ← zero_extend(offset || 02) + GPR[sp]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
for i ← 0 to fn(reglist)
    (pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
    pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
    bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
    datadoubleword ← GPR[gpr(reglist, i)]63-8*bytesel..0 || 08*bytesel
    StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
    vAddr ← vAddr + 4
endfor

```

```
function fn(list)
  fn ← number of entries in list - 1
endfunction
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch



Format: SWP rs1, offset(base)

microMIPS

Purpose: Store Word Pair

To store two consecutive words to memory

Description: $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{GPR}[\text{rs1}], \text{GPR}[\text{rs1}+1]$

The least-significant 32-bit words of GPR *rs1* and GPR *rs1+1* are stored in memory at the location specified by the aligned effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

It is implementation-specific whether interrupts are disabled during the sequence of operations generated by this instruction.

Restrictions:

The effective address must be 32-bit aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

The behavior of the instructions is **UNDEFINED** if *rd* equals \$31.

The behavior of this instruction is **UNDEFINED**, if it is placed in a delay slot of a jump or branch.

Operation:

```

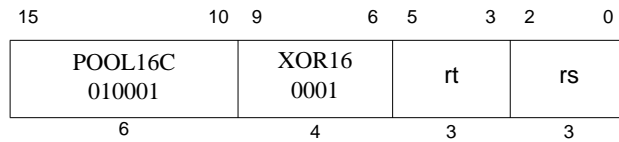
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
datadoubleword ← GPR[rs1]63-8*bytesel..0 || 08*bytesel
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)

vAddr ← sign_extend(offset) + GPR[base] + 4
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
datadoubleword ← GPR[rs1+1]63-8*bytesel..0 || 08*bytesel
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch



Format: XOR16 *rt*, *rs*

microMIPS

Purpose: Exclusive OR (16-bit instr size)

To do a bitwise logical Exclusive OR

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

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical Exclusive OR operation and place the result into GPR *rt*.

Restrictions:

The 3-bit register fields can only specify GPRs \$2-\$7, \$16, \$17.

Operation:

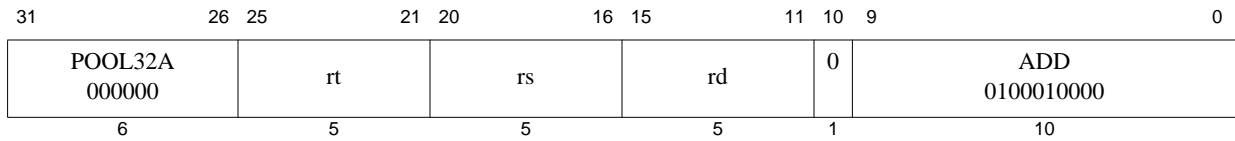
$GPR[rt] \leftarrow GPR[rs] \text{ xor } GPR[rt]$

Exceptions:

None

5.5 Recoded 32-Bit Instructions

This section defines the recoded instructions of the existing MIPS64 instruction sets.



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

microMIPS

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 sign-extended and placed into GPR *rd*.

Restrictions:

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

Operation:

```

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

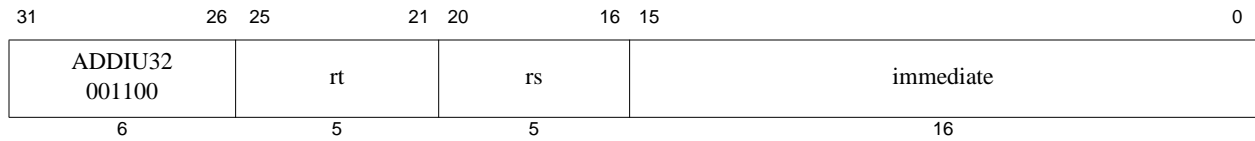
```

Exceptions:

Integer Overflow

Programming Notes:

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



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

microMIPS

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.


```

else
    StoreFPR(fd, PS, ValueFPR(ft, PS)31..0 || ValueFPR(fs,PS)63..32)
endif

```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Programming Notes:

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

```

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

```

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

```

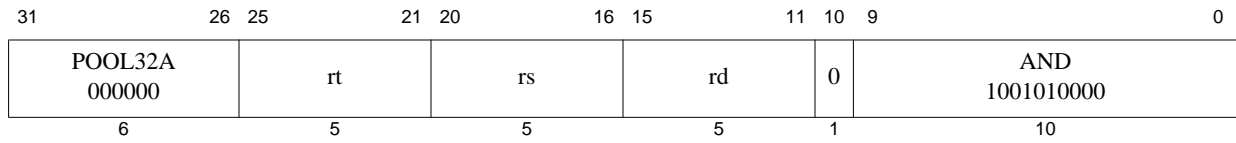
/* T1[i] = T0[i] + F8, T0 aligned, T1 unaligned. */
   CVT.PS.S F8, F8, F8/* make addend paired-single */

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

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

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

```

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

microMIPS

Purpose: And

To do a bitwise logical AND

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

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:

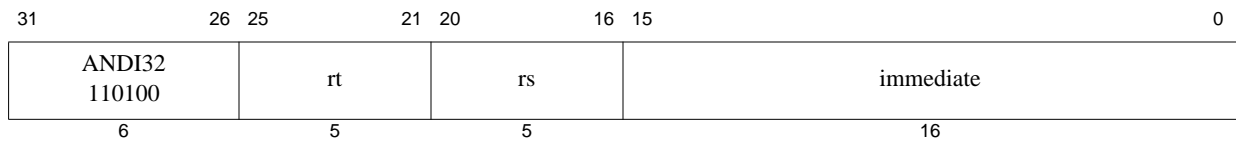
None

Operation:

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

Exceptions:

None



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

microMIPS

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:

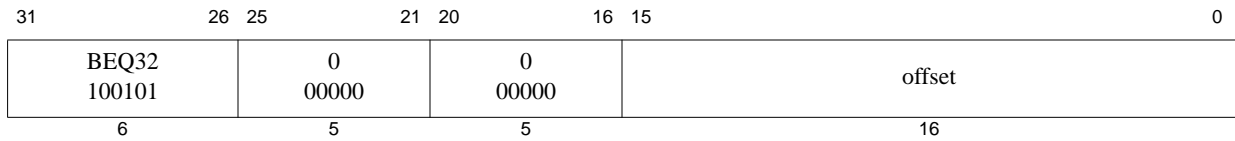
None

Operation:

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

Exceptions:

None



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 17-bit signed offset (the 16-bit *offset* field shifted left 1 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:

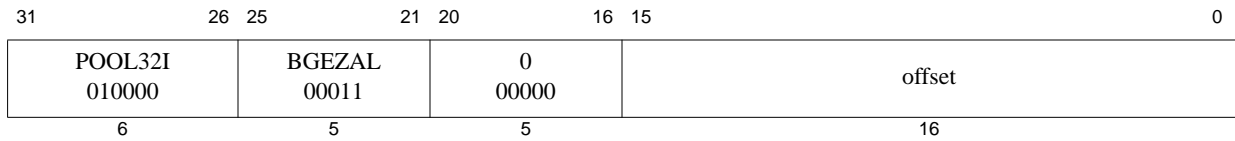
I: target_offset ← sign_extend(offset || 0¹)
I+1: PC ← PC + target_offset

Exceptions:

None

Programming Notes:

With the 17-bit signed instruction offset, the conditional branch range is ± 64 Kbytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.



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 17-bit signed offset (the 16-bit *offset* field shifted left 1 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

Restrictions:

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

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

Operation:

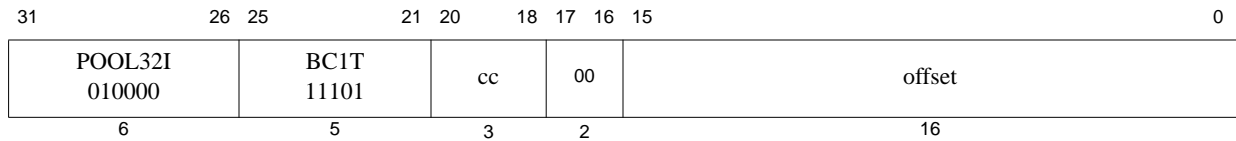
I: $\text{target_offset} \leftarrow \text{sign_extend}(\text{offset} \mid\mid 0^1)$
 $\text{GPR}[31] \leftarrow \text{PC} + 8$
I+1: $\text{PC} \leftarrow \text{PC} + \text{target_offset}$

Exceptions:

None

Programming Notes:

With the 17-bit signed instruction offset, the conditional branch range is ± 64 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.



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

microMIPS
microMIPS

Purpose: Branch on FP True

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

Description: if $FPConditionCode(cc) = 1$ then branch

A 17-bit signed offset (the 16-bit *offset* field shifted left 1 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:

```

I:    condition ← FPConditionCode(cc) = 1
        target_offset ← (offset15)GPRLEN-(16+1) || offset || 01
I+1:  if condition then
        PC ← PC + target_offset
        endif

```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Programming Notes:

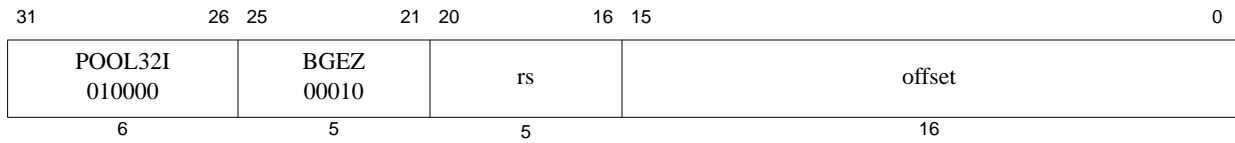
With the 17-bit signed instruction offset, the conditional branch range is ± 64 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

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.



Format: BGEZ *rs*, *offset*

microMIPS

Purpose: Branch on Greater Than or Equal to Zero

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

Description: if $GPR[rs] \geq 0$ then branch

A 17-bit signed offset (the 16-bit *offset* field shifted left 1 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 || 01)
        condition ←  $GPR[rs] \geq 0^{GPRLEN}$ 
I+1:  if condition then
        PC ← PC + target_offset
        endif

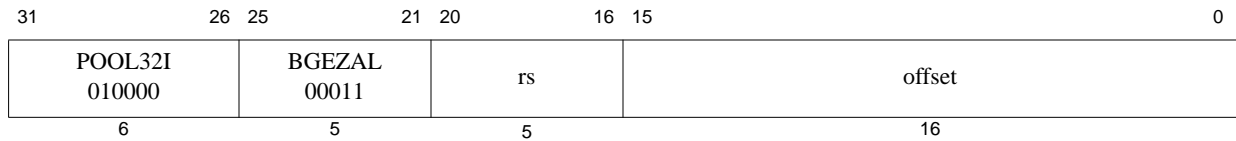
```

Exceptions:

None

Programming Notes:

With the 17-bit signed instruction offset, the conditional branch range is ± 64 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.



Format: BGEZAL *rs*, *offset*

microMIPS

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

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

Description: if $GPR[rs] \geq 0$ 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.

A 17-bit signed offset (the 16-bit *offset* field shifted left 1 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:

The delay-slot instruction must be 32-bits in size. Processor operation is **UNPREDICTABLE** if a 16-bit instruction is placed in the delay slot of BGEZAL.

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 || 01)
        condition ← GPR[rs] ≥ 0GPRLEN
        GPR[31] ← PC + 8
I+1: if condition then
        PC ← PC + target_offset
        endif

```

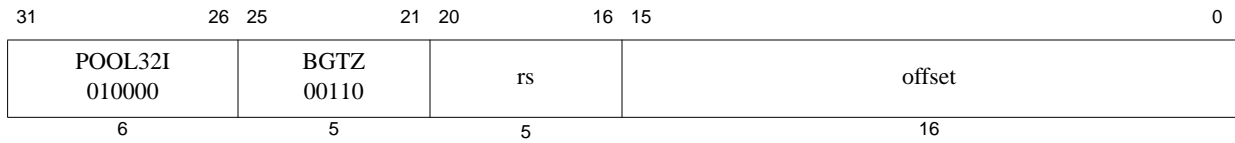
Exceptions:

None

Programming Notes:

With the 17-bit signed instruction offset, the conditional branch range is ± 64 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

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



Format: BGTZ *rs*, *offset*

microMIPS

Purpose: Branch on Greater Than Zero

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

Description: if $GPR[rs] > 0$ then branch

A 17-bit signed offset (the 16-bit *offset* field shifted left 1 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 || 01)
        condition ← GPR[rs] > 0GPRLEN
I+1:  if condition then
        PC ← PC + target_offset
        endif

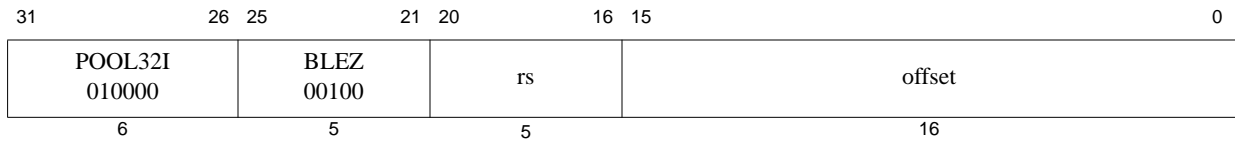
```

Exceptions:

None

Programming Notes:

With the 17-bit signed instruction offset, the conditional branch range is ± 64 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.



Format: BLEZ *rs*, *offset*

microMIPS

Purpose: Branch on Less Than or Equal to Zero

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

Description: if $GPR[rs] \leq 0$ then branch

A 17-bit signed offset (the 16-bit *offset* field shifted left 1 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 || 01)
        condition ← GPR[rs] ≤ 0GPRLEN
I+1:  if condition then
        PC ← PC + target_offset
        endif

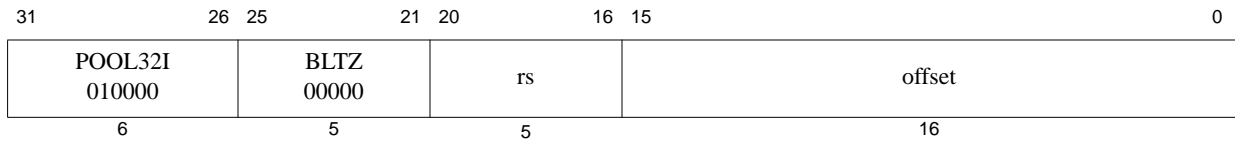
```

Exceptions:

None

Programming Notes:

With the 17-bit signed instruction offset, the conditional branch range is ± 64 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.



Format: BLTZ *rs*, *offset*

microMIPS

Purpose: Branch on Less Than Zero

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

Description: if $GPR[rs] < 0$ then branch

A 17-bit signed offset (the 16-bit *offset* field shifted left 1 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 || 01)
        condition ← GPR[rs] < 0GPRLEN
I+1:  if condition then
        PC ← PC + target_offset
        endif

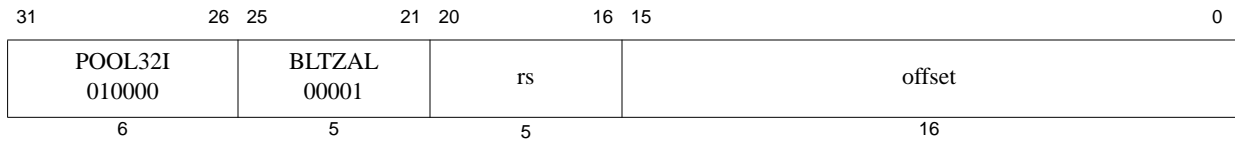
```

Exceptions:

None

Programming Notes:

With the 17-bit signed instruction offset, the conditional branch range is ± 64 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.



Format: BLTZAL *rs*, *offset*

microMIPS

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.

A 17-bit signed offset (the 16-bit *offset* field shifted left 1 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:

The delay-slot instruction must be 32-bits in size. Processor operation is **UNPREDICTABLE** if a 16-bit instruction is placed in the delay slot of BLTZAL.

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 || 01)
        condition ← GPR[rs] < 0GPRLEN
        GPR[31] ← PC + 8
I+1: if condition then
        PC ← PC + target_offset
        endif

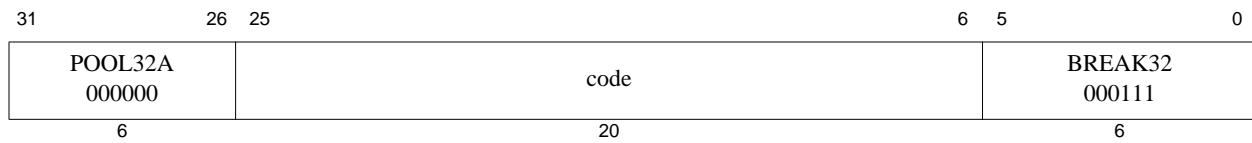
```

Exceptions:

None

Programming Notes:

With the 17-bit signed instruction offset, the conditional branch range is ± 64 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.



Format: BREAK

microMIPS

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

can be made with Branch on FP False (BC1F).

Table 3.26 shows another set of eight compare operations, distinguished by a *cond₃* 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 5.17 FPU Comparisons Without Special Operand Exceptions

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

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

Table 5.18 FPU Comparisons With Special Operand Exceptions for QNaNs

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

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

Restrictions:

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

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

The result of C.cond.PS is **UNPREDICTABLE** if the processor is executing in the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU,.

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

Operation:

```

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

```

```

    unordered ← false
endif
condition ← (cond2 and less) or (cond1 and equal)
            or (cond0 and unordered)
SetFPConditionCode(cc, condition)

```

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

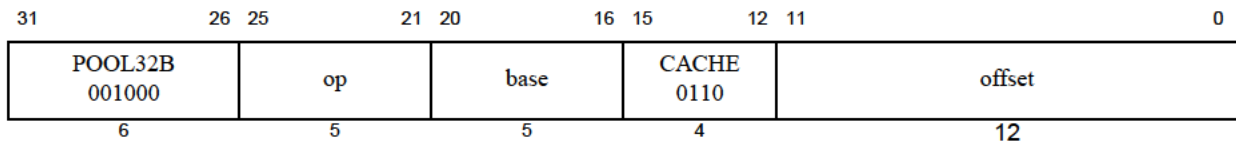
Programming Notes:

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

```

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

```

Format: CACHE op, offset(base)

microMIPS

Purpose: Perform Cache Operation

To perform the cache operation specified by op.

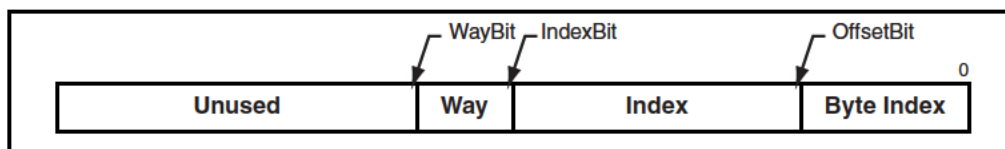
Description:

The 12-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 5.19 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> $\text{OffsetBit} \leftarrow \text{Log}_2(\text{BPT})$ $\text{IndexBit} \leftarrow \text{Log}_2(\text{CS} / \text{A})$ $\text{WayBit} \leftarrow \text{IndexBit} + \text{Ceiling}(\text{Log}_2(\text{A}))$ $\text{Way} \leftarrow \text{Addr}_{\text{WayBit}-1..\text{IndexBit}}$ $\text{Index} \leftarrow \text{Addr}_{\text{IndexBit}-1..\text{OffsetBit}}$ <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 5.2 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 a 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 [22:21] of the instruction specify the cache on which to perform the operation, as follows:

Table 5.20 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 [25:23] 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 cannot properly deal with.

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

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

Table 5.21 Encoding of Bits [20:18] of the CACHE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b000	I	Index Invalidate	Index	Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.	Required
	D	Index Writeback Invalidate / Index Invalidate	Index	For a write-back cache: If the state of the cache block at the specified index is valid and dirty, write the block back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid.	Required
	S, T	Index Writeback Invalidate / Index Invalidate	Index	For a write-through cache: Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. Note that Index Store Tag should be used to initialize the cache at power up.	Required if S, T cache is 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

Table 5.21 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 <i>Hit_Invalidate_D</i> 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.	Required
	S, T	Hit Writeback Invalidate / Hit Invalidate	Address	For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache. 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 5.21 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	<p>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.</p> <p>In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.</p>	Recommended
	S, T	Hit Writeback	Address		Optional, if Hit_Writeback_D is implemented, the S and T variants are recommended.
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 uncacheable.

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

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

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

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

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

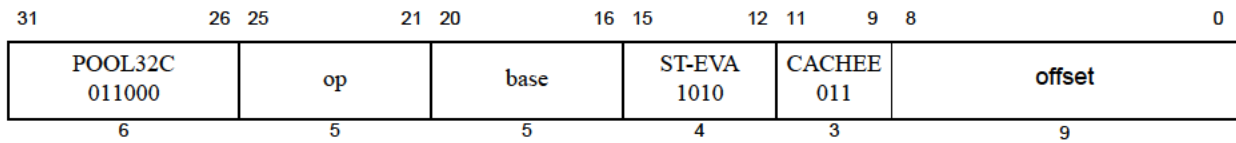
Cache Error Exception

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 */
```

Format: CACHEE op, offset(base)

microMIPS

Purpose: Perform Cache Operation EVA

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

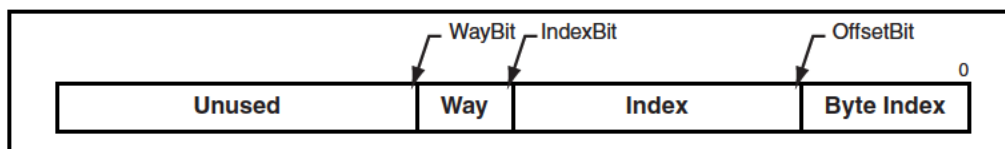
Description:

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 5.22 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> $\text{OffsetBit} \leftarrow \text{Log}_2(\text{BPT})$ $\text{IndexBit} \leftarrow \text{Log}_2(\text{CS} / \text{A})$ $\text{WayBit} \leftarrow \text{IndexBit} + \text{Ceiling}(\text{Log}_2(\text{A}))$ $\text{Way} \leftarrow \text{Addr}_{\text{WayBit}-1..\text{IndexBit}}$ $\text{Index} \leftarrow \text{Addr}_{\text{IndexBit}-1..\text{OffsetBit}}$ <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 5.3 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 a 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 [22:21] of the instruction specify the cache on which to perform the operation, as follows:

Table 5.23 Encoding of Bits[22:21] of CACHEE Instruction

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

Bits [25:23] of the instruction specify the operation to perform. To provide software with a consistent base of cache operations, certain encodings must be supported on all processors. The remaining encodings are recommended

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

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

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

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

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

Table 5.24 Encoding of Bits [20:18] of the CACHEE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b000	I	Index Invalidate	Index	Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.	Required
	D	Index Writeback Invalidate / Index Invalidate	Index	For a write-back cache: If the state of the cache block at the specified index is valid and dirty, write the block back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid.	Required
	S, T	Index Writeback Invalidate / Index Invalidate	Index	For a write-through cache: Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. Note that Index Store Tag should be used to initialize the cache at power up.	Required if S, T cache is implemented

Table 5.24 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	<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.</p> <p>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
0b010	All	Index Store Tag	Index	<p>Write the tag for the cache block at the specified index from the <i>TagLo</i> and <i>TagHi</i> Coprocessor 0 registers. This operation must not cause a Cache Error Exception.</p> <p>This required encoding may be used by software to initialize the entire instruction or data caches by stepping through all valid indices. Doing so requires that the <i>TagLo</i> and <i>TagHi</i> registers associated with the cache be initialized first.</p>	Required
0b011	All	Implementation Dependent	Unspecified	Available for implementation-dependent operation.	Optional
0b100	I, D	Hit Invalidate	Address	<p>If the cache block contains the specified address, set the state of the cache block to invalid.</p> <p>This required encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache.</p>	<p>Required (Instruction Cache Encoding Only), Recommended otherwise</p> <p>Optional, if <i>Hit_Invalidate_D</i> is implemented, the S and T variants are recommended.</p>
	S, T	Hit Invalidate	Address	<p>In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.</p>	

Table 5.24 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.	Required
	S, T	Hit Writeback Invalidate / Hit Invalidate	Address	For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache. 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 5.24 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

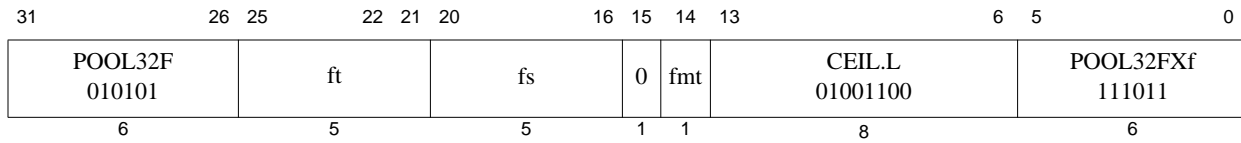
Cache Error Exception

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 */
```



Format: CEIL.L.fmt
 CEIL.L.S ft, fs
 CEIL.L.D ft, fs

microMIPS
 microMIPS

Purpose: Fixed Point Ceiling Convert to Long Fixed Point

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

Description: $FPR[ft] \leftarrow \text{convert_and_round}(FPR[fs])$

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 *ft*.

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

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

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

Operation:

$\text{StoreFPR}(ft, L, \text{ConvertFmt}(\text{ValueFPR}(fs, \text{fmt}), \text{fmt}, L))$

Exceptions:

Coprocessor Unusable, Reserved Instruction

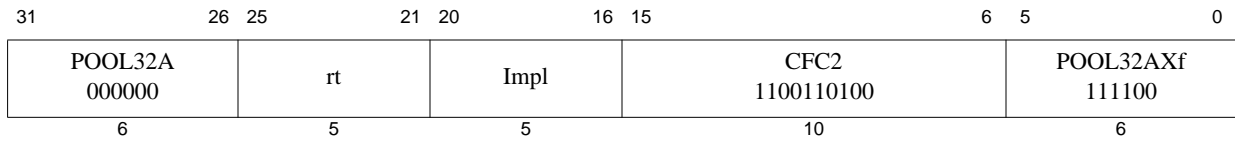
Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

ately following CFC1.

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

MIPS32r5 introduced the UFR and UNFR register aliases that allow user level access to *Status_{FR}*.



Format: CFC2 rt, Impl

microMIPS

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:

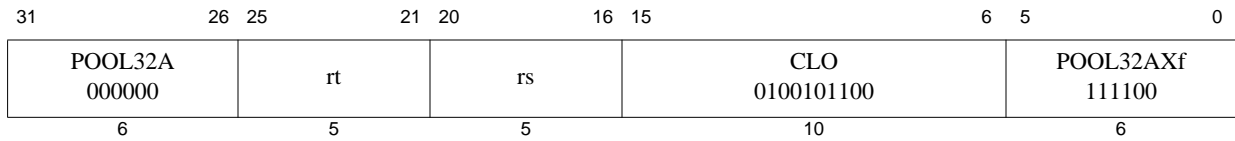
The result is **UNPREDICTABLE** if *Impl* specifies a register that does not exist.

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction



Format: CLO *rt*, *rs*

microMIPS

Purpose: Count Leading Ones in Word

To count the number of leading ones in a word

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

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 *rt*. If all of bits **31..0** were set in GPR *rs*, the result written to GPR *rt* is 32.

Restrictions:

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

Operation:

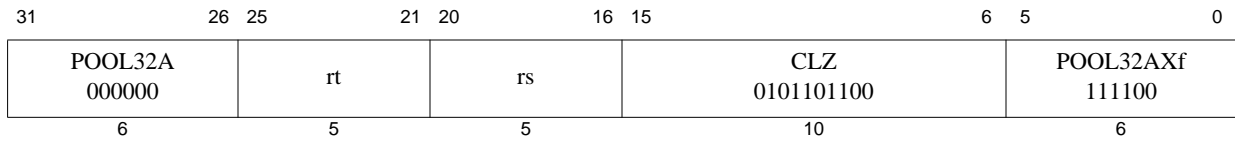
```

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

```

Exceptions:

None



Format: CLZ *rt*, *rs*

microMIPS

Purpose: Count Leading Zeros in Word

Count the number of leading zeros in a word

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

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 *rt*. If no bits were set in GPR *rs*, the result written to GPR *rt* is 32.

Restrictions:

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

Operation:

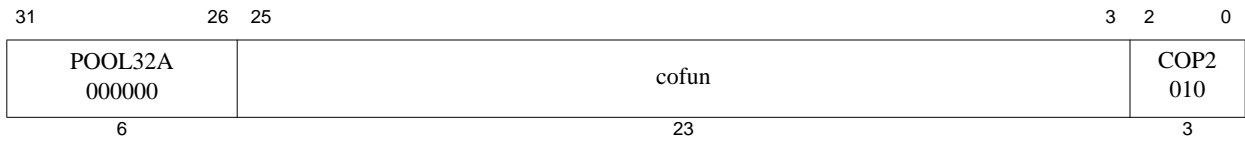
```

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

```

Exceptions:

None



Format: COP2 func

microMIPS

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


```

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

```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

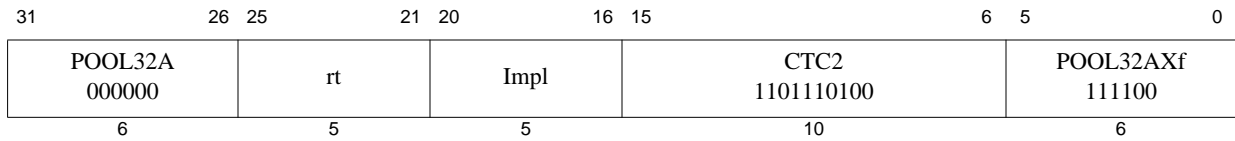
Unimplemented Operation, Invalid Operation, Division-by-zero, Inexact, Overflow, Underflow

Historical Information:

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

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

MIPS32r5 introduced the UFR and UNFR register aliases that allow user level access to *Status_{FR}*.



Format: CTC2 rt, Impl

microMIPS

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:

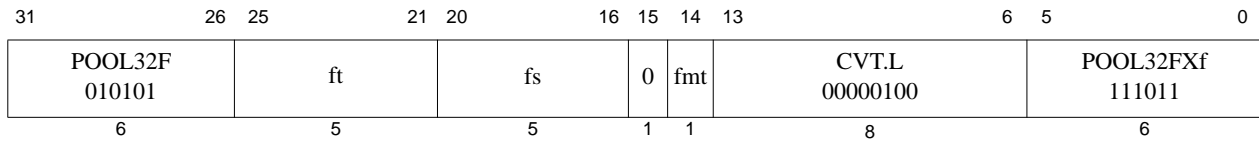
The result is **UNPREDICTABLE** if *rd* specifies a register that does not exist.

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction



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

microMIPS
microMIPS

Purpose: Floating Point Convert to Long Fixed Point

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

Description: $FPR[ft] \leftarrow \text{convert_and_round}(FPR[fs])$

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 *ft*.

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

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

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

Operation:

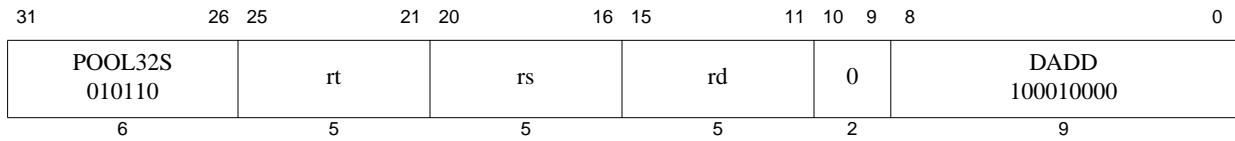
`StoreFPR (ft, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))`

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact,



Format: DADD rd, rs, rt

microMIPS64

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:

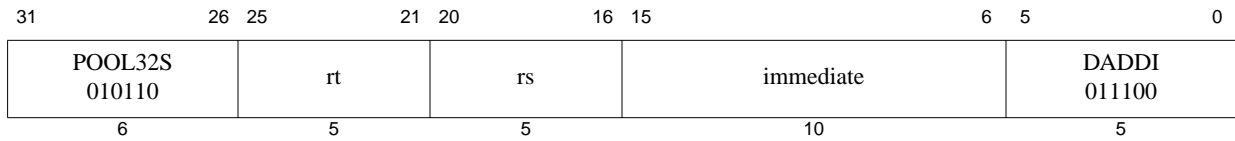
```
temp ← (GPR[rs]63 || GPR[rs]) + (GPR[rt]63 || GPR[rt])
if (temp64 ≠ temp63) then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp63..0
endif
```

Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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



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

microMIPS64

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

Restrictions:

Operation:

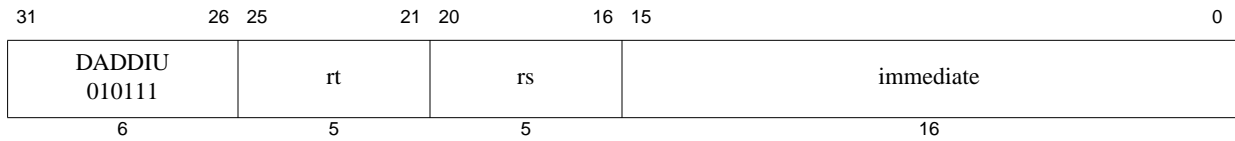
```
temp ← (GPR[rs]63 || GPR[rs]) + sign_extend(immediate)
if (temp64 ≠ temp63) then
    SignalException(IntegerOverflow)
else
    GPR[rt] ← temp63..0
endif
```

Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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



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

microMIPS64

Purpose: Doubleword Add Immediate Unsigned

To add a constant to a 64-bit integer

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

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

No Integer Overflow exception occurs under any circumstances.

Restrictions:

Operation:

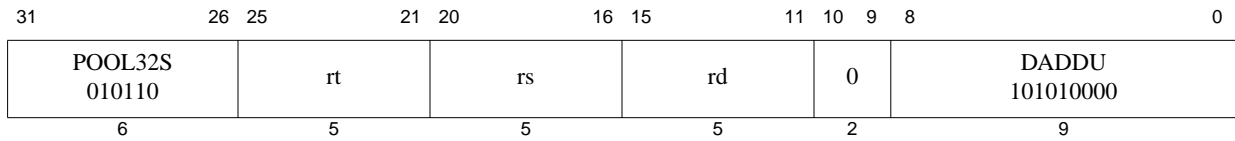
$GPR[rt] \leftarrow GPR[rs] + sign_extend(immediate)$

Exceptions:

Reserved Instruction

Programming Notes:

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



Format: DADDU rd, rs, rt

microMIPS64

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:

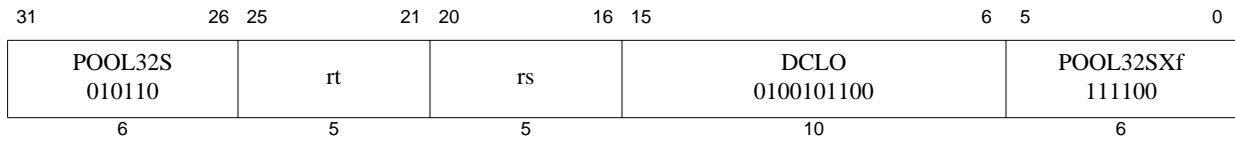
$$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$$

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.



Format: DCLO rd, rs

microMIPS64

Purpose: Count Leading Ones in Doubleword

To count the number of leading ones in a doubleword

Description: $GPR[rd] \leftarrow \text{count_leading_ones } GPR[rs]$

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:

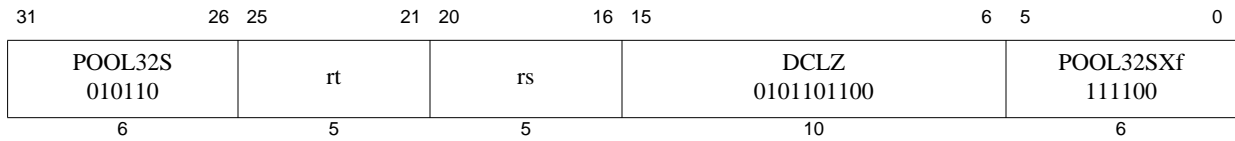
To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the *rt* and *rd* fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values.

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



Format: DCLZ rd, rs

microMIPS64

Purpose: Count Leading Zeros in Doubleword

To count the number of leading zeros in a doubleword

Description: $GPR[rd] \leftarrow \text{count_leading_zeros } GPR[rs]$

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:

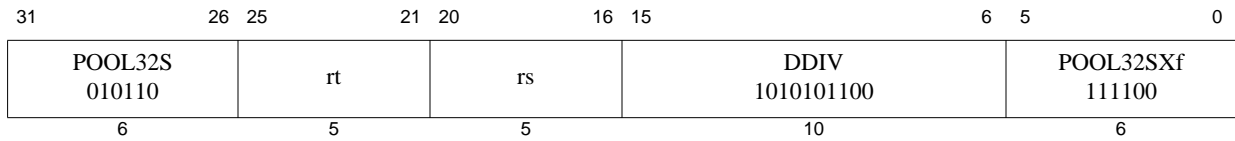
To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the *rt* and *rd* fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values.

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



Format: DDIV *rs*, *rt*

microMIPS64

Purpose: Doubleword Divide

To divide 64-bit signed integers

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

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

No arithmetic exception occurs under any circumstances.

Restrictions:

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

Operation:

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

Exceptions:

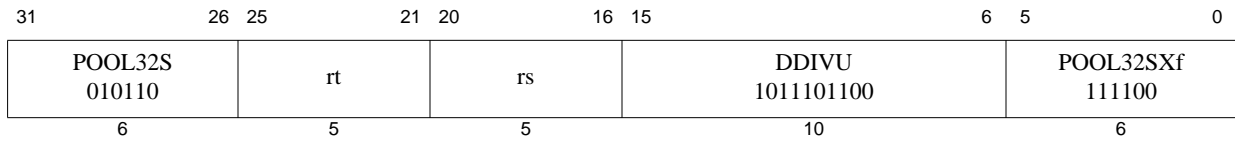
Reserved Instruction

Programming Notes:

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

Historical Perspective:

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



Format: DDIVU rs, rt

microMIPS64

Purpose: Doubleword Divide Unsigned

To divide 64-bit unsigned integers

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

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

No arithmetic exception occurs under any circumstances.

Restrictions:

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

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	16 15	6 5	0
POOL32A 000000	0 0000000000	DERET 1110001101	POOL32AXf 111100	
6	10	10	6	

Format: DERET

EJTAG microMIPS

Purpose: Debug Exception Return

To Return from a debug exception.

Description:

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

Restrictions:

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

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

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

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

Operation:

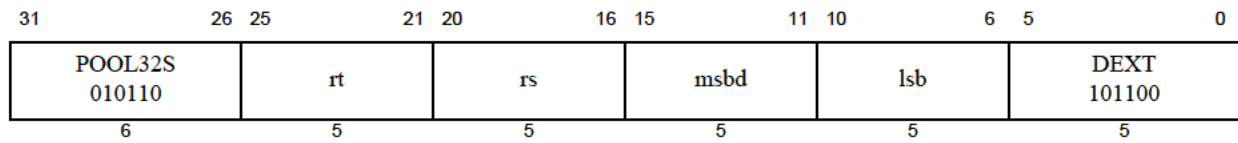
```

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

```

Exceptions:

Coprocessor Unusable Exception
Reserved Instruction Exception



Format: DEXT *rt*, *rs*, *pos*, *size*

microMIPS64

Purpose: Doubleword Extract Bit Field

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

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

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

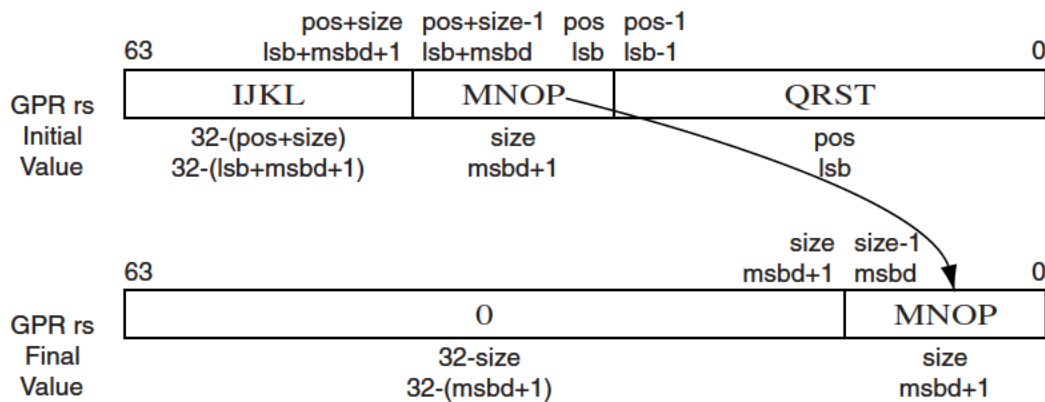
```
msbd ← size-1
lsb ← pos
msb ← lsb+msbd
```

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

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

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

Figure 5.4 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:

<i>msbd</i>	<i>lsb</i>	<i>msb</i>	<i>pos</i>	<i>size</i>	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
$32 \leq msbd < 64$	$0 \leq lsb < 32$	$32 \leq msb < 64$	$0 \leq pos < 32$	$32 < size \leq 64$	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.

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

Operation:

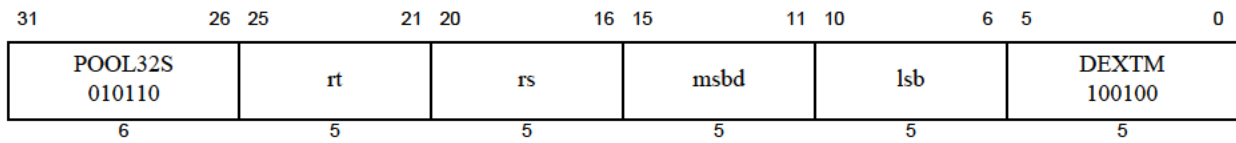
$$\text{GPR}[\text{rt}] \leftarrow 0^{63-(\text{msbd}+1)} \parallel \text{GPR}[\text{rs}]_{\text{msbd}+1:\text{lsb}}$$

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.



Format: DEXTM *rt*, *rs*, *pos*, *size*

microMIPS

Purpose: Doubleword Extract Bit Field Middle

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

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

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

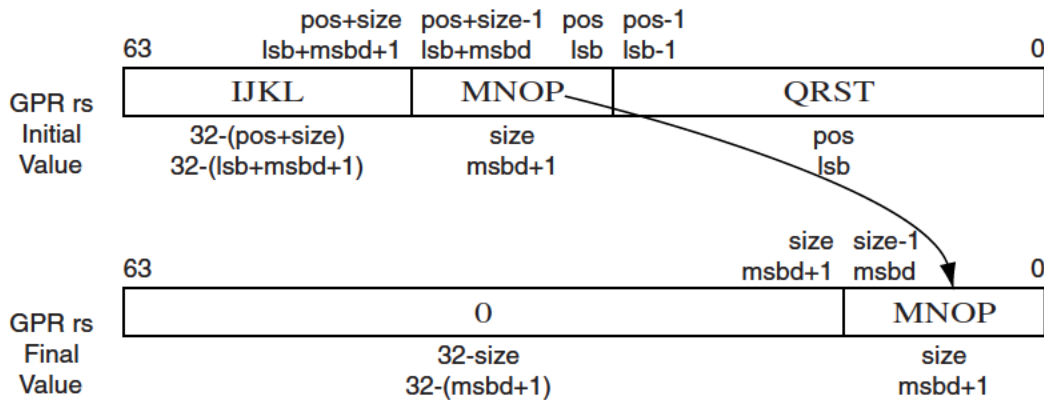
```
msbdminus32 ← size-1-32
lsb ← pos
msbd ← msbdminus32 + 32
msb ← lsb+msbd
```

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

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

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

Figure 5.5 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:

<i>msbd</i>	<i>lsb</i>	<i>msb</i>	<i>pos</i>	<i>size</i>	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
$32 \leq \text{msbd} < 64$	$0 \leq \text{lsb} < 32$	$32 \leq \text{msb} < 64$	$0 \leq \text{pos} < 32$	$32 < \text{size} \leq 64$	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 $(\text{lsb} + \text{msbd} + 1) > 64$.

Operation:

```

msbd ← msbminus32 + 32
if ((lsb + msbd + 1) > 64) then
    UNPREDICTABLE
endif
GPR[rt] ← 063-(msbd+1) || GPR[rs]msbd+lsb..pos

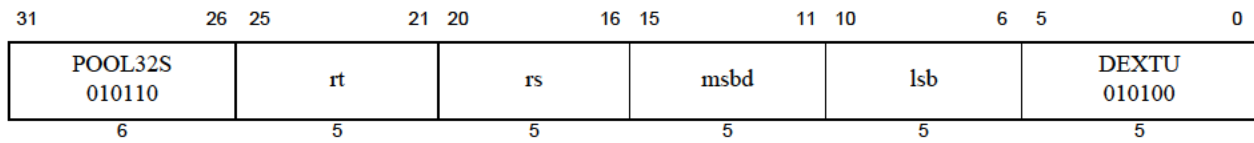
```

Exceptions:

Reserved Instruction

Programming Notes

The assembler will accept any value of *pos* and *size* that satisfies the relationship $0 < \text{pos} + \text{size} \leq 64$ 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.



Format: DEXTU *rt*, *rs*, *pos*, *size*

microMIPS64

Purpose: Doubleword Extract Bit Field Upper

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

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

The bit field starting at bit *pos* and extending for *size* bits is extracted from GPR *rs* and stored zero-extended and right-justified in GPR *rt*. The assembly language arguments *pos* and *size* are converted by the assembler to the instruction fields *msbd* (the most significant bit of the destination field in GPR *rt*), in instruction bits 15..11, and *lsbminus32* (least significant bit of the source field in GPR *rs*, minus 32), in instruction bits 10..6, as follows:

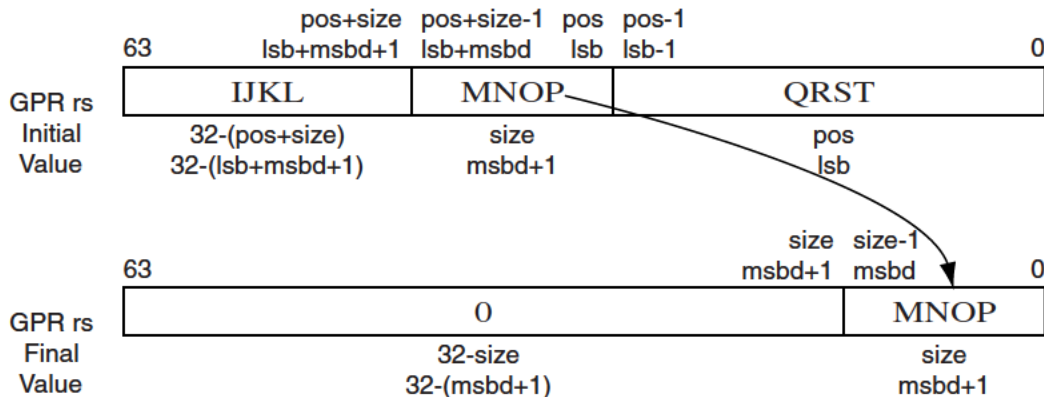
```
msbd ← size-1
lsbminus32 ← pos-32
lsb ← lsbminus32 + 32
msb ← lsb+msbd
```

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 5.6 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:

<i>msbd</i>	<i>lsb</i>	<i>msb</i>	<i>pos</i>	<i>size</i>	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

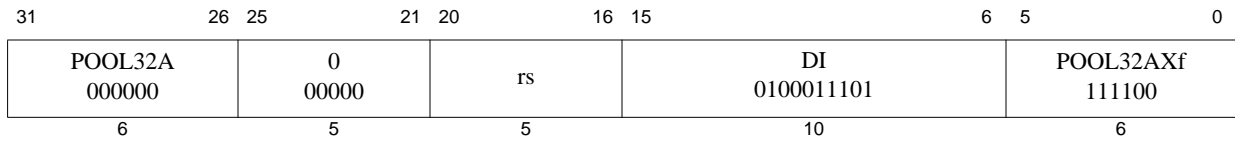
```

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.



Format: DI
DI rs

microMIPS
microMIPS

Purpose: Disable Interrupts

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

Description: $GPR[rs] \leftarrow Status; Status_{IE} \leftarrow 0$

The current value of the *Status* register is sign-extended and loaded into general register *rs*. 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:

```
data ← Status
GPR[rs] ← 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 cannot be aborted in the middle by an interrupt or exception.

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

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.

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:

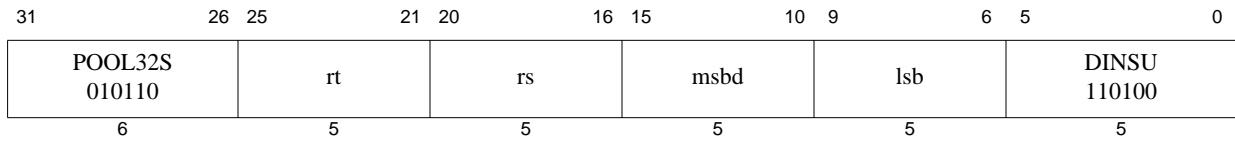
$$msb \leftarrow msb_{\text{minus}32} + 32$$

$$GPR[rt] \leftarrow GPR[rt]_{63..msb+1} \ || \ GPR[rs]_{msb-lsb..0} \ || \ GPR[rt]_{lsb-1..0}$$
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.



Format: DINSU *rt*, *rs*, *pos*, *size*

microMIPS64

Purpose: Doubleword Insert Bit Field Upper

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

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

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
    
```

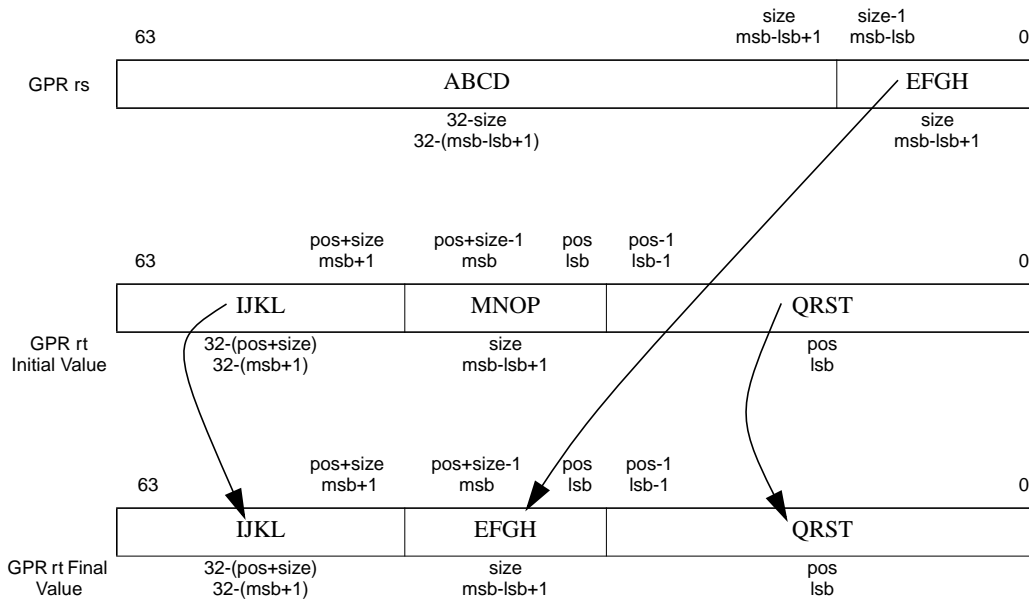
For this instruction, the values of *pos* and *size* must satisfy all of the following relations:

```

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

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

Figure 5.9 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:

```

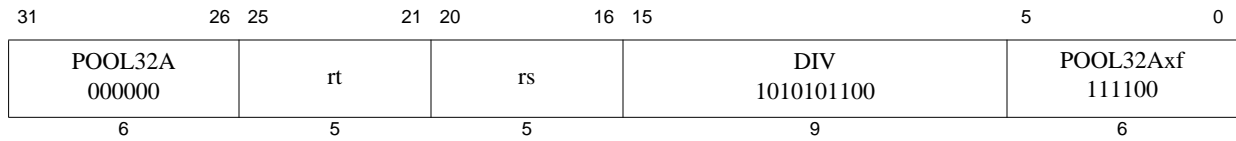
lsb ← lsbminus32 + 32
msb ← msbminus32 + 32
if (lsb > msb) then
    UNPREDICTABLE
endif
GPR[rt] ← GPR[rt]63..msb+1 || GPR[rs]msb-lsb..0 || GPR[rt]lsb-1..0
    
```

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.



Format: DIV *rs*, *rt*

microMIPS

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  $\leftarrow$  GPR[rs]31..0 div GPR[rt]31..0
LO  $\leftarrow$  sign_extend(q31..0)
r  $\leftarrow$  GPR[rs]31..0 mod GPR[rt]31..0
HI  $\leftarrow$  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.

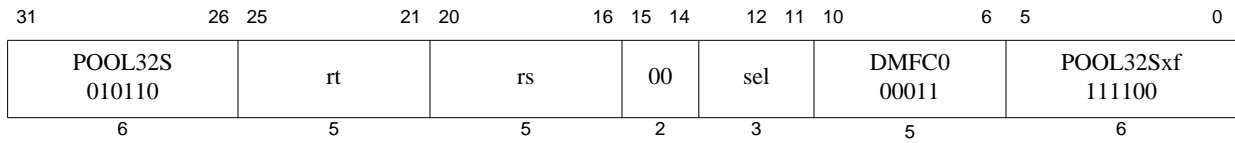
By default, most compilers for the MIPS architecture will emit additional instructions to check for the divide-by-zero and overflow cases when this instruction is used. In many compilers, the assembler mnemonic “DIV r0, rs, rt” can be used to prevent these additional test instructions to be emitted.

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



Format: DMFC0 *rt*, *rs*
 DMFC0 *rt*, *rs*, *sel*

microMIPS64
microMIPS64

Purpose: Doubleword Move from Coprocessor 0

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

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

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:

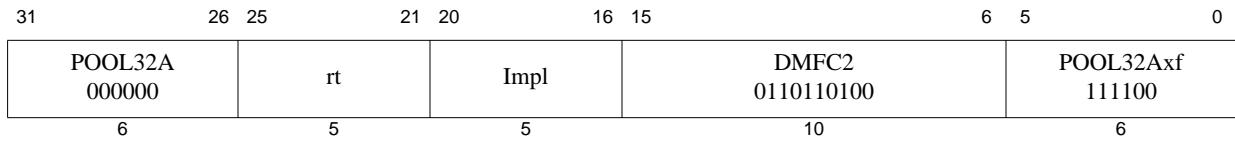
The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *rs* and *sel*, or if the coprocessor 0 register specified by *rd* and *sel* is a 32-bit register.

Operation:

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

Exceptions:

- Coprocessor Unusable
- Reserved Instruction



Format: DMFC2 *rt*, *rd* **microMIPS64**
 DMFC2, *rt*, *rd*, *sel* **microMIPS64**

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:

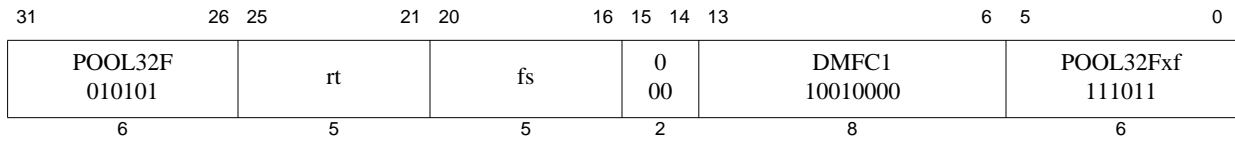
The results are **UNPREDICTABLE** if *impl* specifies a coprocessor 2 register that does not exist, or if the coprocessor 2 register specified by *rd* and *sel* is a 32-bit register.

Operation:

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

Exceptions:

- Coprocessor Unusable
- Reserved Instruction



Format: DMFC1 *rt*, *fs*

microMIPS64

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

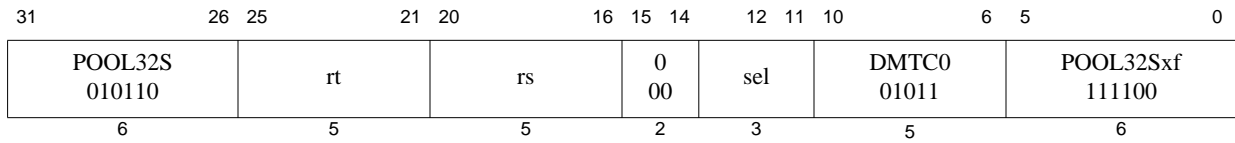
Exceptions:

Coprocessor Unusable

Reserved Instruction

Historical Information:

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



Format: DMTC0 *rt*, *rs*
 DMTC0 *rt*, *rs*, *sel*

microMIPS64
microMIPS64

Purpose: Doubleword Move to Coprocessor 0

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

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

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 *rs* and *sel*, or if the coprocessor 0 register specified by *rd* and *sel* is a 32-bit register.

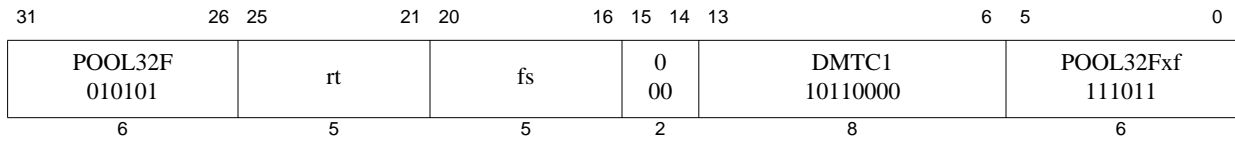
Operation:

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

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: DMTC1 rt, fs

microMIPS64

Purpose: Doubleword Move to Floating Point

To copy a doubleword from a GPR to an FPR

Description: FPR[fs] ← GPR[rt]

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

Restrictions:

Operation:

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

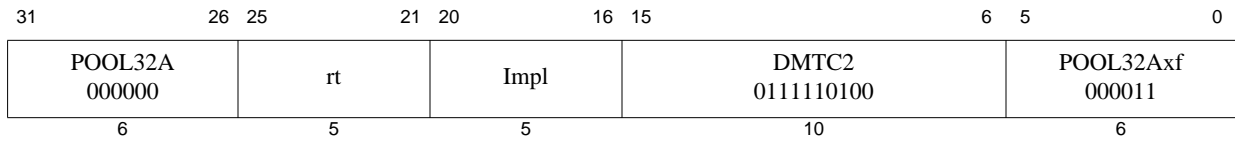
Exceptions:

Coprocessor Unusable

Reserved Instruction

Historical Information:

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



Format: DMTC2 rt, Impl **microMIPS64**
 DMTC2 rt, Impl, sel **microMIPS64**

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:

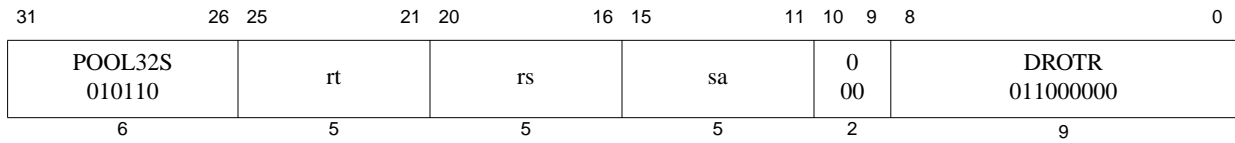
The results are **UNPREDICTABLE** if *Impl* specifies a coprocessor 2 register that does not exist, or if the coprocessor 2 register specified by *rd* and *sel* is a 32-bit register.

Operation:

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

Exceptions:

- Coprocessor Unusable
- Reserved Instruction



Format: DROTR *rt, rs, sa*

microMIPS64

Purpose: Doubleword Rotate Right

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

Description: $GPR[rt] \leftarrow GPR[rs] \leftrightarrow (\text{right})\ sa$

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

Restrictions:

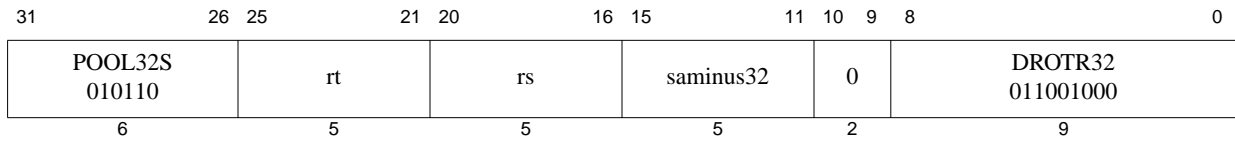
Operation:

$$s \leftarrow 0 \ ||\ sa$$

$$GPR[rt] \leftarrow GPR[rs]_{s-1..0} \ ||\ GPR[rs]_{63..s}$$

Exceptions:

Reserved Instruction



Format: DROTR32 *rt*, *rs*, *sa*

microMIPS64

Purpose: Doubleword Rotate Right Plus 32

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

Description: $GPR[rt] \leftarrow GPR[rs] \leftrightarrow (\text{right}) (saminus32+32)$

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

Restrictions:

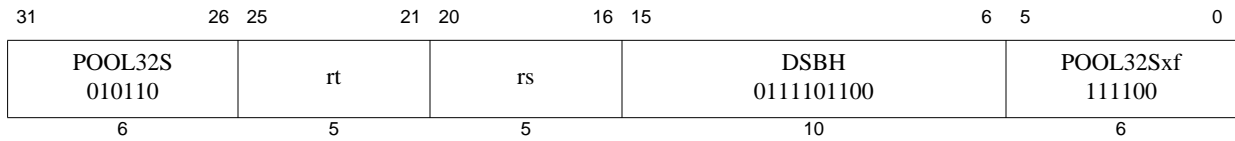
Operation:

$$s \leftarrow 1 \mid \mid sa \quad /* 32+saminus32 */$$

$$GPR[rt] \leftarrow GPR[rs]_{s-1..0} \mid \mid GPR[rt]_{63..s}$$

Exceptions:

Reserved Instruction



Format: DSBH *rt*, *rs*

microMIPS64

Purpose: Doubleword Swap Bytes Within Halfwords

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

Description: $GPR[rt] \leftarrow \text{SwapBytesWithinHalfwords}(GPR[rs])$

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

Restrictions:

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

Operation:

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

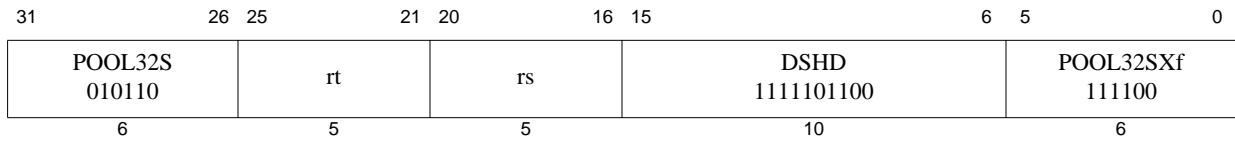
Exceptions:

Reserved Instruction

Programming Notes:

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

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

Format: DSHD *rt*, *rs*

microMIPS64

Purpose: Doubleword Swap Halfwords Within Doublewords

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

Description: $GPR[rt] \leftarrow \text{SwapHalfwordsWithinDoublewords}(GPR[rs])$

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

Restrictions:

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

Operation:

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

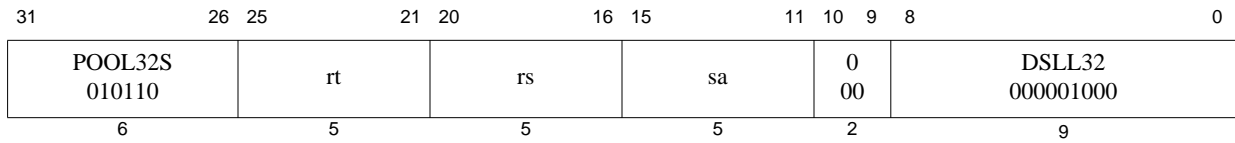
Exceptions:

Reserved Instruction

Programming Notes:

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

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

Format: DSLL32 *rt*, *rs*, *sa*

microMIPS64

Purpose: Doubleword Shift Left Logical Plus 32

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

Description: $GPR[rt] \leftarrow GPR[rs] \ll (sa+32)$

The 64-bit doubleword contents of GPR *rs* are shifted left, inserting zeros into the emptied bits; the result is placed in GPR *rt*. The bit-shift amount in the range 0 to 31 is specified by *sa*.

Restrictions:

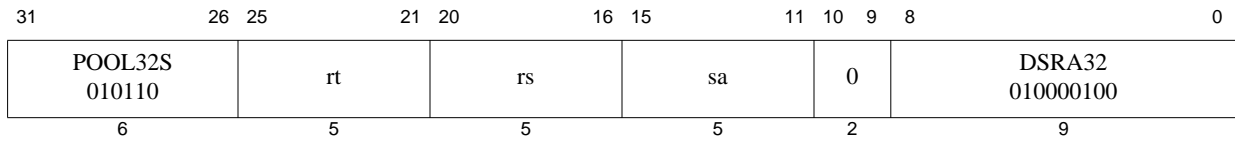
Operation:

$$GPR[rt] \leftarrow 1 \ll sa \quad /* \ 32+sa \ */$$

$$GPR[rt] \leftarrow GPR[rs]_{(63-s)..0} \ll 0^s$$

Exceptions:

Reserved Instruction



Format: DSRA32 *rt*, *rs*, *sa*

microMIPS64

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: $GPR[rt] \leftarrow GPR[rs] \gg (sa+32)$ (arithmetic)

The doubleword contents of GPR *rs* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR *rt*. The bit-shift amount in the range 32 to 63 is specified by *sa*+32.

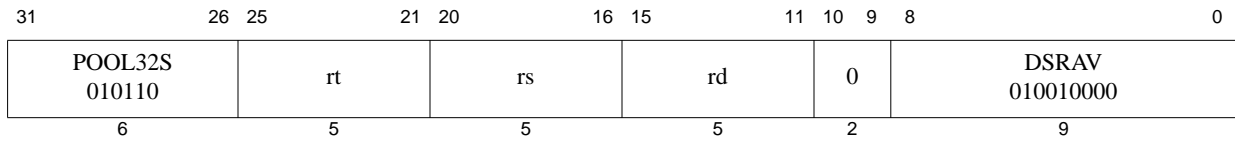
Restrictions:

Operation:

$$GPR[rt] \leftarrow (GPR[rs]_{63})^s \parallel GPR[rs]_{63..s}$$

Exceptions:

Reserved Instruction



Format: DSRAV rd, rt, rs

microMIPS64

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*.

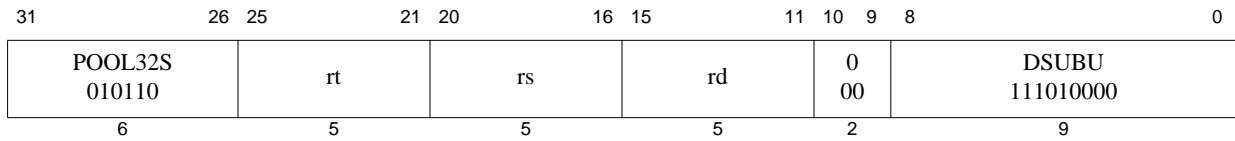
Restrictions:

Operation:

$$GPR[rd] \leftarrow (GPR[rt]_{63})^s \parallel GPR[rt]_{63..s}$$

Exceptions:

Reserved Instruction



Format: DSUBU rd, rs, rt

microMIPS64

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

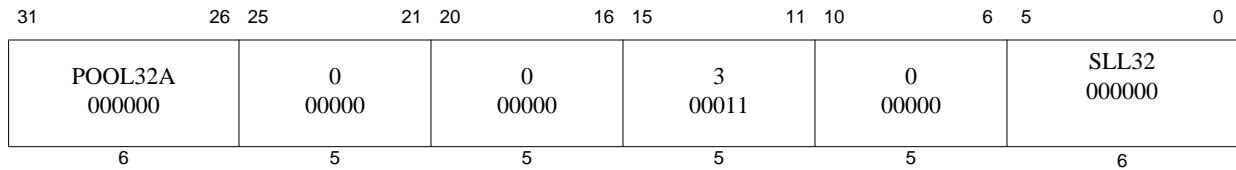
$$GPR[rd] \leftarrow GPR[rs] - GPR[rt]$$

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.



Format: EHB

microMIPS

Purpose: Execution Hazard Barrier

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

Description:

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

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

Restrictions:

None

Operation:

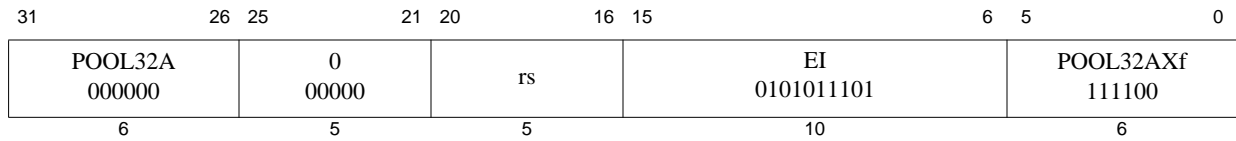
```
ClearExecutionHazards()
```

Exceptions:

None

Programming Notes:

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



Format: EI
EI rs

microMIPS
microMIPS

Purpose: Enable Interrupts

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

Description: $GPR[rt] \leftarrow Status; Status_{IE} \leftarrow 1$

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:

```

data ← Status
GPR[rs] ← 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 cannot be aborted in the middle by an interrupt or exception.

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

31	26 25	16 15	6 5	0
POOL32A 000000	0 0000000000	ERET 1111001101	POOL32AXf 111100	
6	10	10	6	

Format: ERET

microMIPS

Purpose: Exception Return

To return from interrupt, exception, or error trap.

Description:

ERET clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ in a Release 2 implementation, and returns to the interrupted instruction at the completion of interrupt, exception, or error processing. ERET does not execute the next instruction (i.e., it has no delay slot).

Restrictions:

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

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

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

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

Operation:

```

if 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	16	15	6	5	0
POOL32A 000000	0 000000000		1	ERET 1111001101		POOL32AXf 111100	
6	9		1	10		6	

Format: ERETNC

microMIPS Release 5

Purpose: Exception Return No Clear

To return from interrupt, exception, or error trap without clearing the LLbit.

Description:

ERETNC clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ when implemented, and returns to the interrupted instruction at the completion of interrupt, exception, or error processing. ERETNC does not execute the next instruction (i.e., it has no delay slot).

ERETNC is identical to ERET except that an ERETNC will not clear the LLbit that is set by execution of an LL instruction, and thus when placed between an LL and SC sequence, will never cause the SC to fail.

An ERET should continue to be used by default in interrupt and exception processing handlers: the handler may have accessed a synchronizable block of memory common to code that is atomically accessing the memory, and where the code caused the exception or was interrupted. Similarly, a process context-swap must also continue to use an ERET in order to avoid a possible false success on execution of SC in the restored context.

Multiprocessor systems with non-coherent cores (i.e., without hardware coherence snooping) should also continue to use ERET, since it is the responsibility of software to maintain data coherence in the system.

An ERETNC is useful in cases where interrupt/exception handlers and kernel code involved in a process context-swap can guarantee no interference in accessing synchronizable memory across different contexts. ERETNC can also be used in an OS-level debugger to single-step through code for debug purposes, avoiding the false clearing of the LLbit and thus failure of an LL and SC sequence in single-stepped code.

Software can detect the presence of ERETNC by reading $Config5_{LLB}$.

Restrictions:

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

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

Operation:

```

if 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
ClearHazards()
```

Exceptions:

Coprocessor Unusable Exception


```
UNPREDICTABLE  
endif  
temp ← sign_extend( $0^{32-(msbd+1)}$  || GPR[rs]msbd+1sb..1sb)  
GPR[rt] ← temp
```

Exceptions:

Reserved Instruction

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

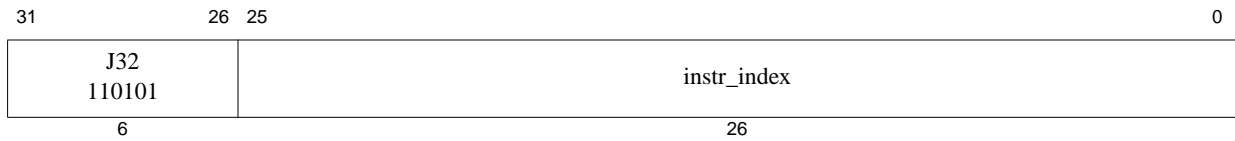
If either GPR rs or GPR rt does not contain sign-extended 32-bit values (bits **63..31** equal), then the result of the operation is **UNPREDICTABLE**.

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]lsb-1..0)
```

Exceptions:

Reserved Instruction



Format: J target

microMIPS

Purpose: Jump

To branch within the current 128 MB-aligned region

Description:

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 128 MB-aligned region. The low 27 bits of the target address is the *instr_index* field shifted left 1 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

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

Restrictions:

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

Operation:

I:
I+1: $PC \leftarrow PC_{\text{GPRLLEN}-1..27} \parallel \text{instr_index} \parallel 0^1$

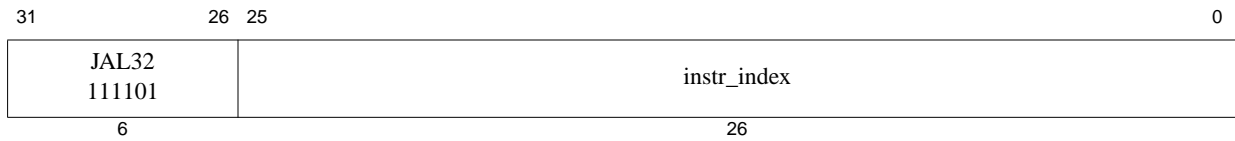
Exceptions:

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 128 MB region aligned on a 128 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the jump instruction is in the last word of a 128 MB region, it can branch only to the following 128 MB region containing the branch delay slot.



Format: JAL target

microMIPS

Purpose: Jump and Link

To execute a procedure call within the current 128 MB-aligned region

Description:

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

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 128 MB-aligned region. The low 27 bits of the target address is the *instr_index* field shifted left 1 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

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

Restrictions:

The delay-slot instruction must be 32-bits in size. Processor operation is **UNPREDICTABLE** if a 16-bit instruction is placed in the delay slot of JAL.

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

Operation:

I: GPR[31] ← PC + 8
I+1: PC ← PC_{GPREN-1..27} || instr_index || 0¹

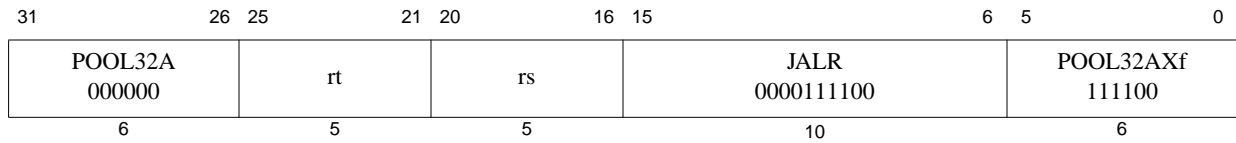
Exceptions:

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 128 MB region aligned on a 128 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 128 MB region, it can branch only to the following 128 MB region containing the branch delay slot.



Format: JALR rs (rt = 31 implied)
JALR rt, rs

microMIPS
microMIPS

Purpose: Jump and Link Register

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

Description: $GPR[rt] \leftarrow return_addr$, $PC \leftarrow GPR[rs]$

Place the return address link in GPR *rt*. 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 MIPS32/64ISA:

- 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 MIPS32/64ISA:

- 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.

Restrictions:

The delay-slot instruction must be 32-bits in size. Processor operation is **UNPREDICTABLE** if a 16-bit instruction is placed in the delay slot of JALR.

Register specifiers *rs* and *rt* 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 of GPR *rs*.

For processors which implement MIPS32/64 and if the ISAMode bit of the target is MIPS32/64 (bit 0 of GPR *rs* is 0) and address bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

For processors that do not implement MIPS32/64 ISA, if the intended target ISAMode is MIPS32/64 (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

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

Operation:

```

I: temp ← GPR[rs]
    GPR[rt] ← PC + 8
I+1: if Config1CA = 0 then
    PC ← temp
    else

```

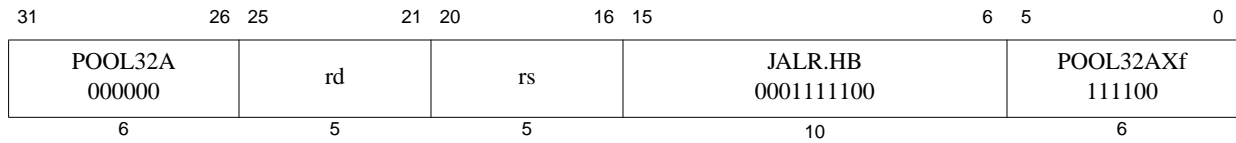
```
PC ← tempGPRLEN-1..1 || 0  
ISAMode ← temp0  
endif
```

Exceptions:

None

Programming Notes:

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



Format: JALR.HB *rs* (*rt* = 31 implied)
JALR.HB *rt*, *rs*

microMIPS
microMIPS

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: $GPR[rt] \leftarrow return_addr$, $PC \leftarrow GPR[rs]$, clear execution and instruction hazards

Place the return address link in GPR *rt*. 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 MIPS32/64 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 MIPS32/64 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.

Restrictions:

The delay-slot instruction must be 32-bits in size. Processor operation is **UNPREDICTABLE** if a 16-bit instruction is placed in the delay slot of JAL.HB.

Register specifiers *rs* and *rd* must not be equal, 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.

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 which implement MIPS32/64 and if the ISAMode bit of the target address is MIPS32/64 (bit 0 of GPR *rs* is 0) and address bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

For processors that do not implement MIPS32/64 ISA, if the intended target ISAMode is MIPS32/64 (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is 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, JALRS.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[rt] ← PC + 8
I+1: if Config1_CA = 0 then
      PC ← temp
      else
      PC ← temp_GPRLEN-1..1 || 0
      ISAMode ← temp_0
      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 *rt*, if omitted in the assembly language instruction, is GPR 31.

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

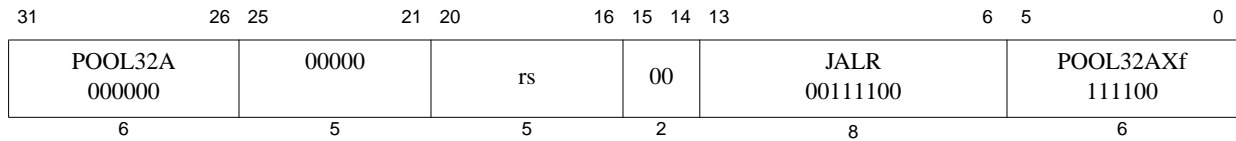
Example: Clearing hazards due to an ASID change

```

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

```

nop



Format: JR *rs*

microMIPS

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 MIPS32/64 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:

The delay-slot instruction must be 32-bits in size. Processor operation is **UNPREDICTABLE** if a 16-bit instruction is placed in the delay slot of JALR.

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 which implement MIPS32/64 and the *ISAMode* bit of the target address is MIPS32/64 (bit 0 of GPR *rs* is 0) and address bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

For processors that do not implement MIPS32/64 ISA, if the intended target *ISAMode* is MIPS32/64 (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

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

Operation:

```

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

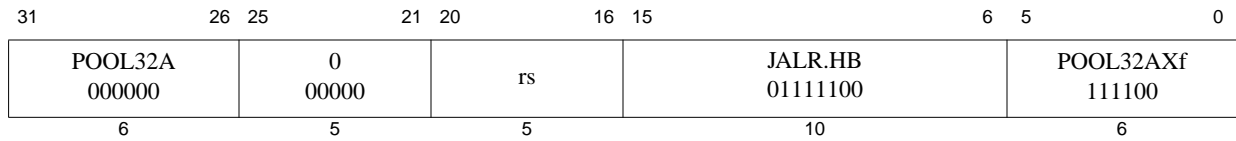
```

Exceptions:

None

Programming Notes:

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



Format: JR.HB rs

microMIPS

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.

For processors that implement the MIPS32/64 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:

The delay-slot instruction must be 32-bits in size. Processor operation is **UNPREDICTABLE** if a 16-bit instruction is placed in the delay slot of JALR.

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 which implement MIPS32/64 and the ISAMode bit of the target address is MIPS32/64 (bit 0 of GPR *rs* is 0) and address bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

For processors that do not implement MIPS32/64 ISA, if the intended target ISAMode is MIPS32/64 (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is 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, JALRS.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 ← GPR[rs]
I+1: if Config1CA = 0 then
    PC ← temp
```

```

else
    PC ← tempGPRLLEN-1..1 || 0
    ISAMode ← temp0
endif
ClearHazards()

```

Exceptions:

None

Programming Notes:

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

Example: Clearing hazards due to an ASID change

```

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

```

Example: Making a write to the instruction stream visible

```

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

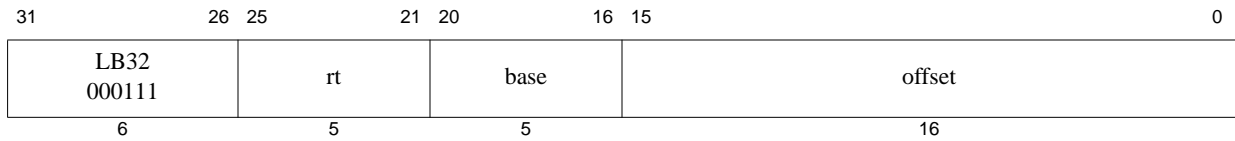
```

Example: Clearing instruction hazards in-line

```

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

```

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

microMIPS

Purpose: Load Byte

To load a byte from memory as a signed value

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

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

Operation:

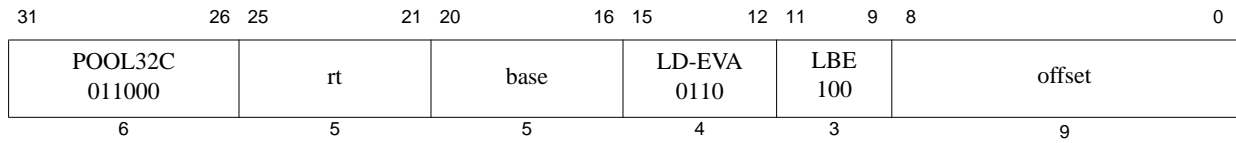
```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor ReverseEndian3)
memdoubleword ← LoadMemory(CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_2..0 xor BigEndianCPU3
GPR[rt] ← sign_extend(memdoubleword7+8*byte..8*byte)

```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch



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

microMIPS

Purpose: Load Byte EVA

To load a byte as a signed value from user mode virtual address space when executing in kernel mode.

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

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

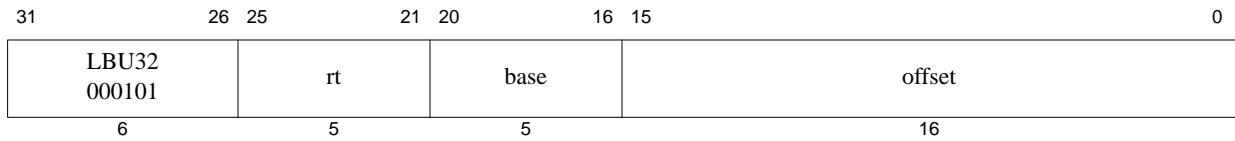
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



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

microMIPS

Purpose: Load Byte Unsigned

To load a byte from memory as an unsigned value

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

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:

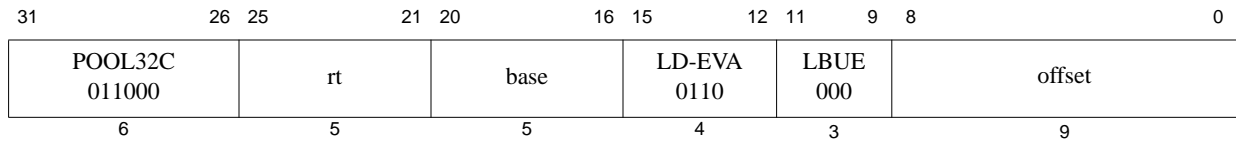
```

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



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

microMIPS

Purpose: Load Byte Unsigned EVA

To load a byte as an unsigned value from user mode virtual address space when executing in kernel mode.

Description: $GPR[rt] \leftarrow memory[GPR[base] + offset]$

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR *rt*. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

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

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

Restrictions:

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

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
pAddr ← 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

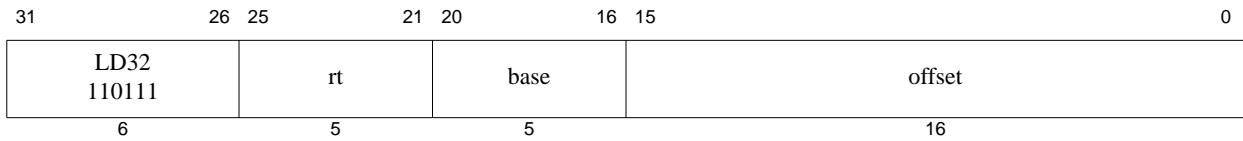
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



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

microMIPS64

Purpose: Load Doubleword

To load a doubleword from memory

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

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

Restrictions:

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

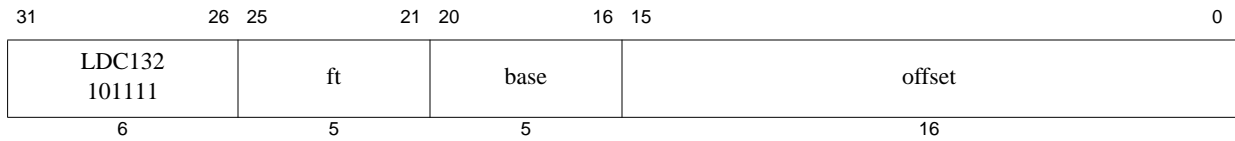
```

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

```

Exceptions:

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



Format: LDC1 ft, offset(base)

microMIPS

Purpose: Load Doubleword to Floating Point

To load a doubleword from memory to an FPR

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

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

Restrictions:

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

Operation:

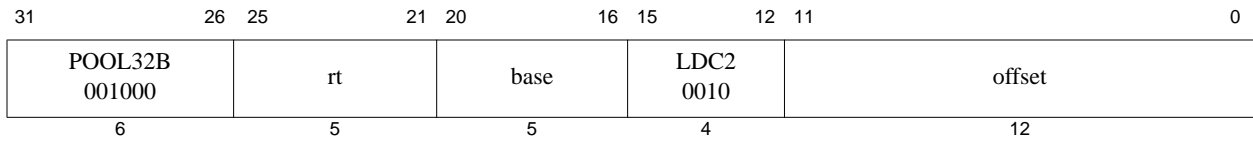
```

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

```

Exceptions:

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



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

microMIPS

Purpose: Load Doubleword to Coprocessor 2

To load a doubleword from memory to a Coprocessor 2 register

Description: $CPR[2,rt,0] \leftarrow \text{memory}[GPR[base] + \text{offset}]$

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in Coprocessor 2 register *rt*. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

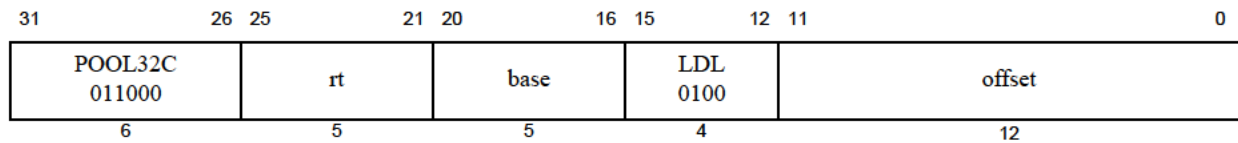
An Address Error exception occurs if $\text{EffectiveAddress}_{2..0} \neq 0$ (not doubleword-aligned).

Operation:

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

Exceptions:

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



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

microMIPS64

Purpose: Load Doubleword Left

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

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

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 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 5.12 Unaligned Doubleword Load Using LDL and LDR

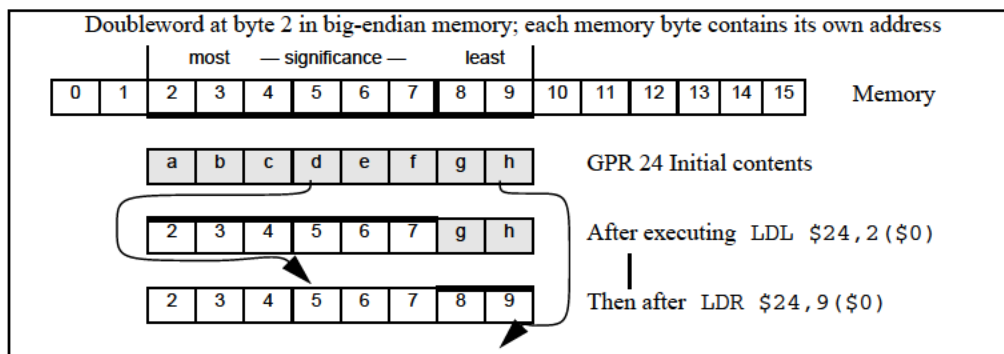
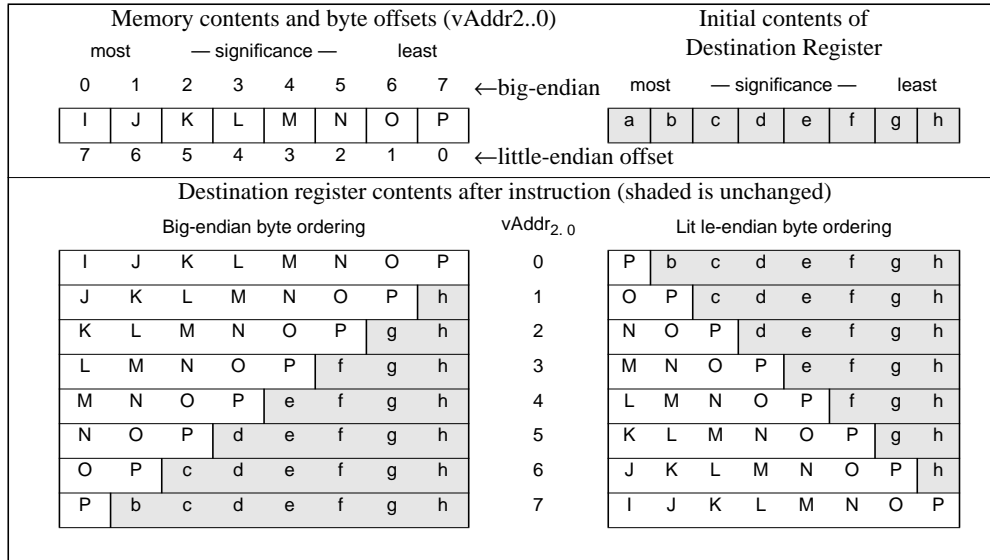


Figure 3-11 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 6 bytes, is located in the aligned doubleword starting with the most-significant byte at 2. LDL first loads these 6 bytes into the 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 5.13 Bytes Loaded by LDL Instruction

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

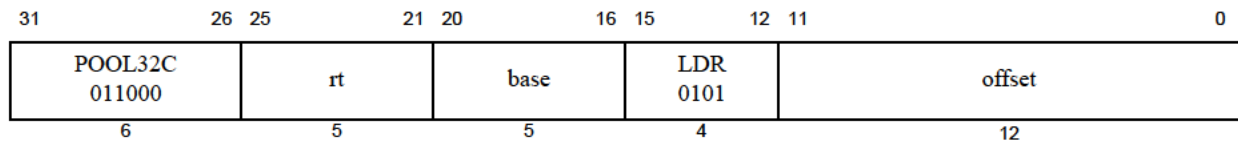
```

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] ← memdoubleworde7+8*byte..0 || GPR[rt]55-8*byte..0

```

Exceptions:

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



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

microMIPS

Purpose: Load Doubleword Right

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

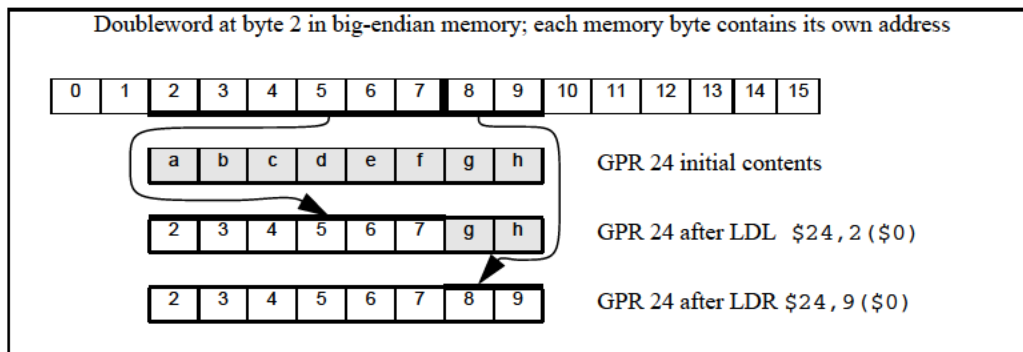
Description: $GPR[rt] \leftarrow GPR[rt] \text{ MERGE } \text{memory}[GPR[base] + \text{offset}]$

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of 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.

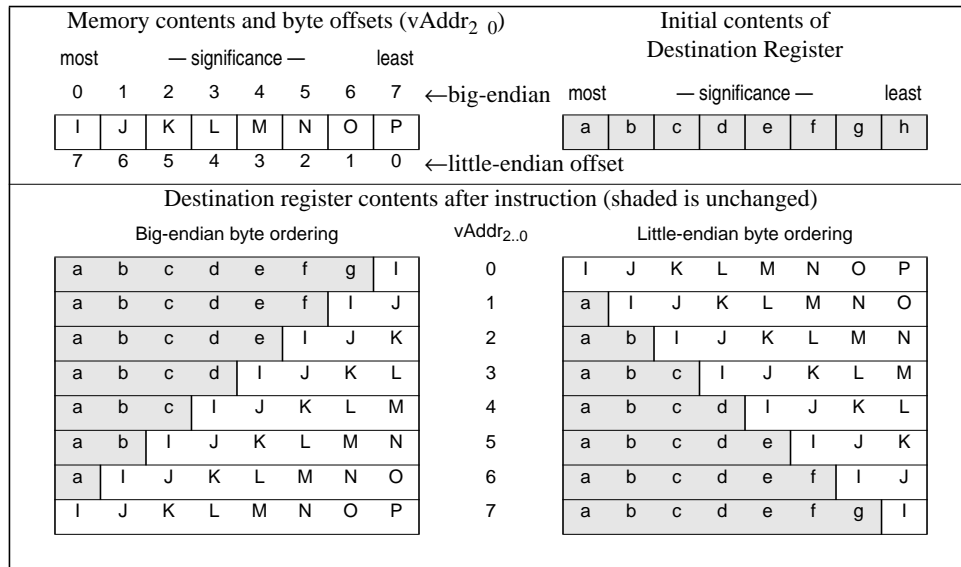
Figure 5.14 Unaligned Doubleword Load Using LDR and LDL



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 5.15 Bytes Loaded by LDR Instruction

**Restrictions:****Operation: 64-bit processors**

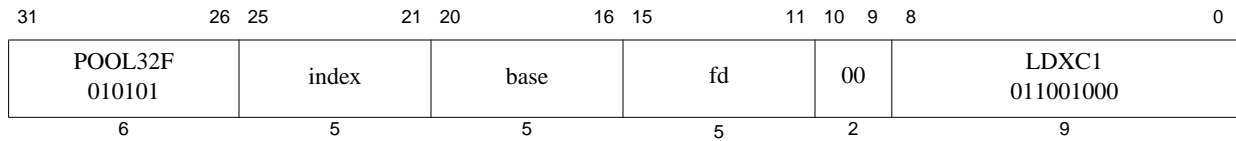
```

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



Format: LDXC1 fd, index(base)

microMIPS
microMIPS

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).

Compatibility and Availability:

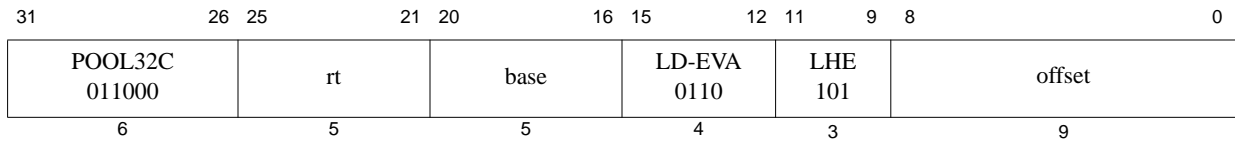
LDXC1: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required in MIPS32r2 and all subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode ($FIR_{F64}=0$ or 1, $Status_{FR}=0$ or 1).

Operation:

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

Exceptions:

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



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

microMIPS

Purpose: Load Halfword EVA

To load a halfword as a signed value from user mode virtual address space when executing in kernel mode.

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

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, and placed in GPR *rt*. The 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 ← 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] ← sign_extend(memdoubleword15+8*byte..8*byte)
```

Exceptions:

TLB Refill

TLB Invalid

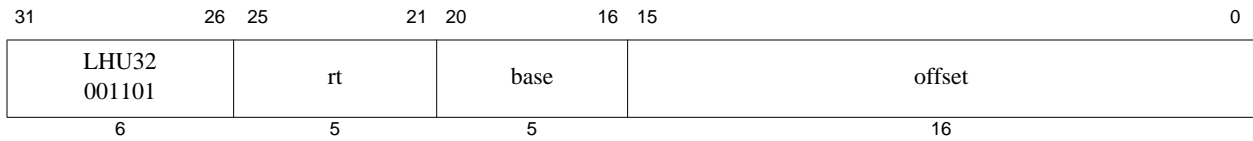
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



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

microMIPS

Purpose: Load Halfword Unsigned

To load a halfword from memory as an unsigned value

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

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:

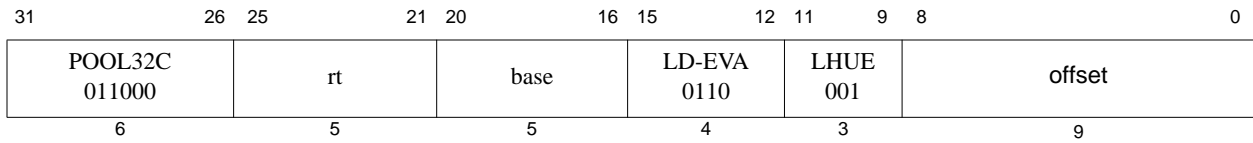
```

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



Format: LHUE rt, offset(base)

microMIPS

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: $GPR[rt] \leftarrow memory[GPR[base] + offset]$

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR *rt*. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

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

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

Restrictions:

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

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

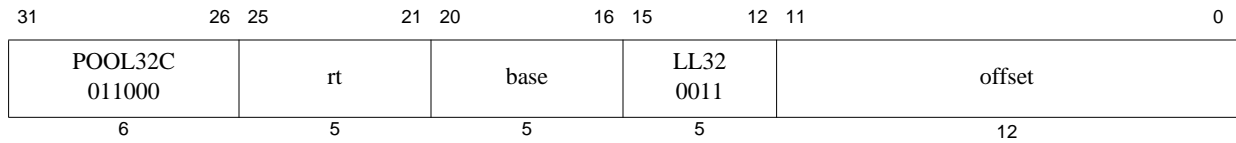
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



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

microMIPS

Purpose: Load Linked Word

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

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

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

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length, and written into GPR *rt*. The 12-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

This begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor. When an LL is executed it starts an active RMW sequence replacing any other sequence that was active. The RMW sequence is completed by a subsequent SC instruction that either completes the RMW sequence atomically and succeeds, or does not and fails.

Executing LL on one processor does not cause an action that, by itself, causes an SC for the same block to fail on another processor.

An execution of LL does not have to be followed by execution of SC; a program is free to abandon the RMW sequence without attempting a write.

Restrictions:

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

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

Operation:

```

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

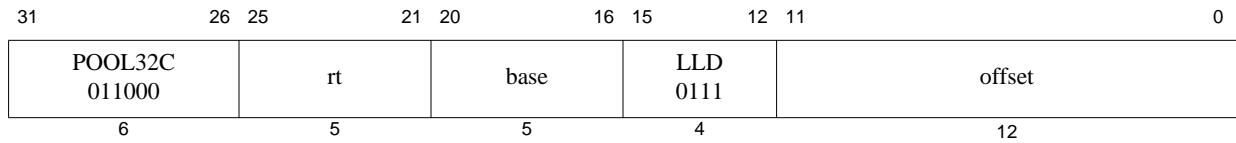
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

Programming Notes:

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



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

microMIPS64

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 in **UNPREDICTABLE**. Which storage is synchronizable is a function of both CPU and system implementations. See the documentation of the **SCD** instruction for the formal definition.

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

Operation:

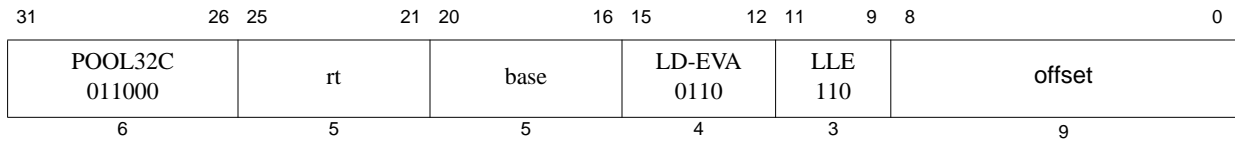
```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[rt] ← memdoubleword
LLbit ← 1

```

Exceptions:

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



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

microMIPS

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

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 12-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)

```

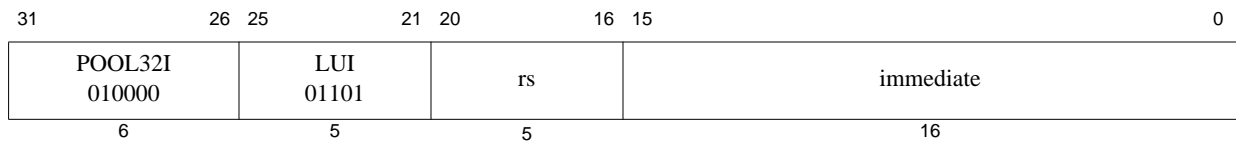
LLbit \leftarrow 1

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.



Format: LUI *rs*, *immediate*

microMIPS

Purpose: Load Upper Immediate

To load a constant into the upper half of a word

Description: $GPR[rs] \leftarrow immediate \parallel 0^{16}$

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

Restrictions:

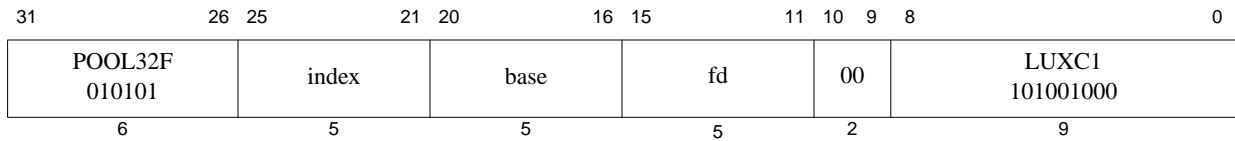
None

Operation:

$GPR[rs] \leftarrow \text{sign_extend}(immediate \parallel 0^{16})$

Exceptions:

None



Format: LUXC1 fd, index(base)

microMIPS
microMIPS

Purpose: Load Doubleword Indexed Unaligned to Floating Point

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

Description: $FPR[fd] \leftarrow memory[(GPR[base] + GPR[index])_{PSIZE-1..3}]$

The contents of the 64-bit doubleword at the memory location specified by the effective address are fetched and placed into the low word of FPR *fd*. The contents of GPR *index* and GPR *base* are added to form the effective address. The effective address is doubleword-aligned; EffectiveAddress_{2..0} are ignored.

Restrictions:

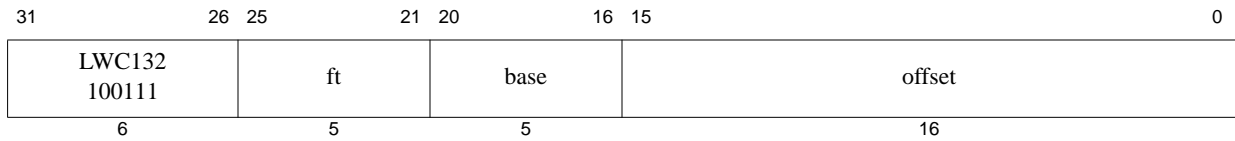
The result of this instruction is **UNPREDICTABLE** if the processor is executing in the *FR=0* 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR=1* mode, but not with *FR=0*, and not on a 32-bit FPU.

Operation:

```
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



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

microMIPS

Purpose: Load Word to Floating Point

To load a word from memory to an FPR

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

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 $\text{EffectiveAddress}_{1..0} \neq 0$ (not word-aligned).

Operation:

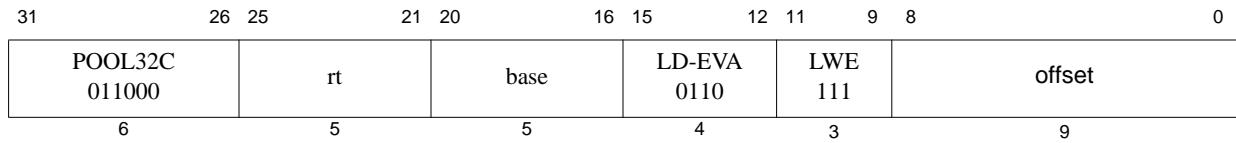
```

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



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

microMIPS

Purpose: Load Word EVA

To load a word from user mode virtual address space when executing in kernel mode.

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 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

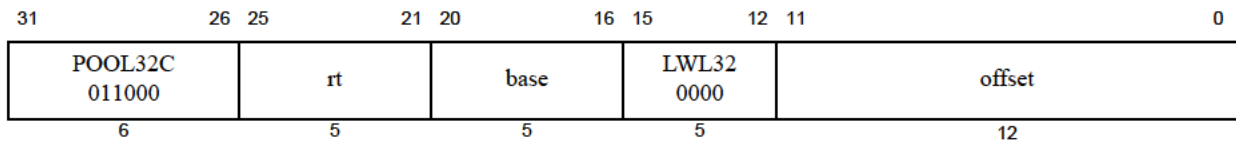
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



Format: LWL rt, offset(base)

microMIPS

Purpose: Load Word Left

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

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

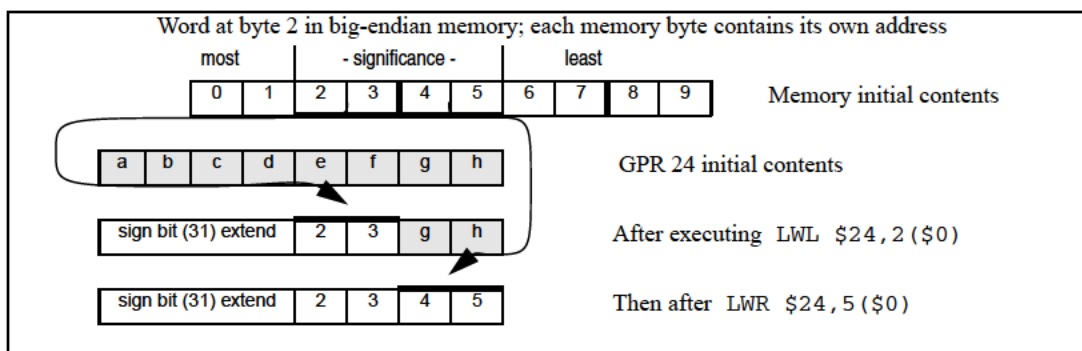
The 12-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

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.

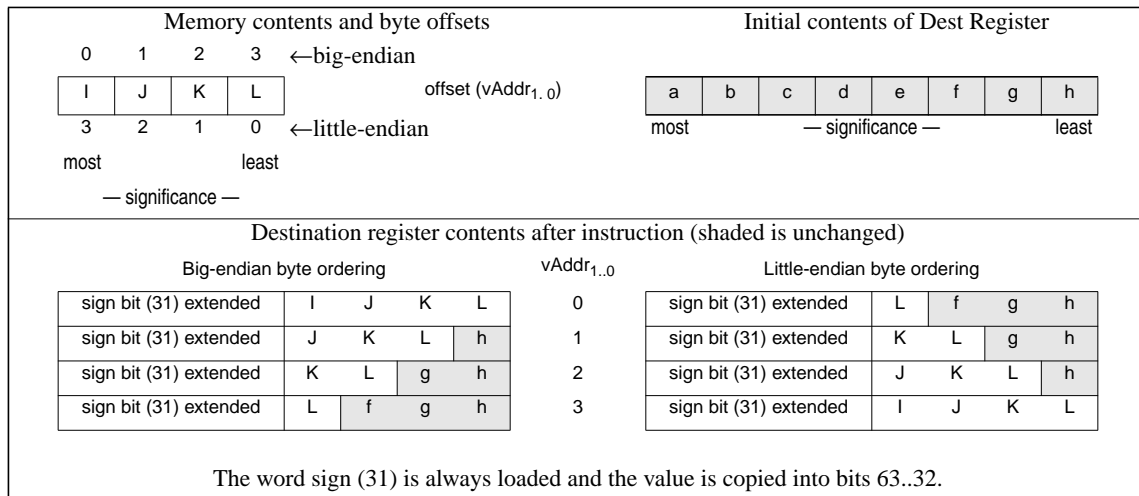
The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2 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 5.16 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 5.17 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

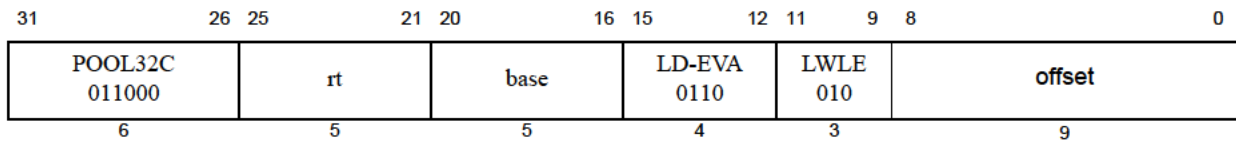
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

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.



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

microMIPS

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

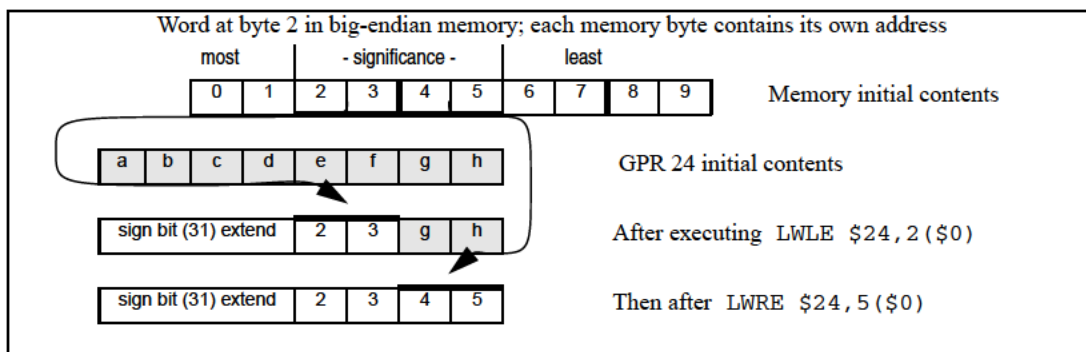
The 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

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 5.18 Unaligned Word Load Using LWLE and LWRE

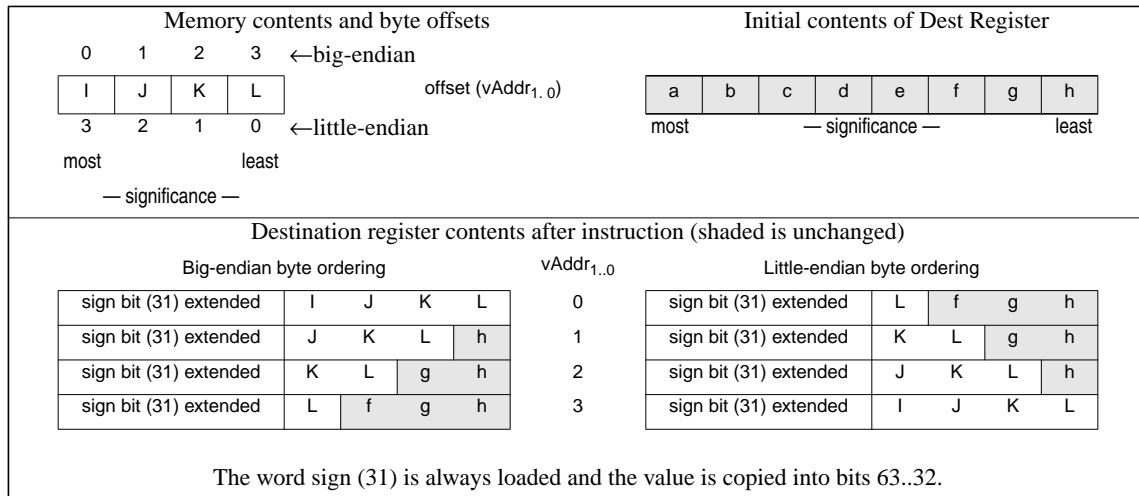


The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address ($vAddr_{1..0}$), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded 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 5.19 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

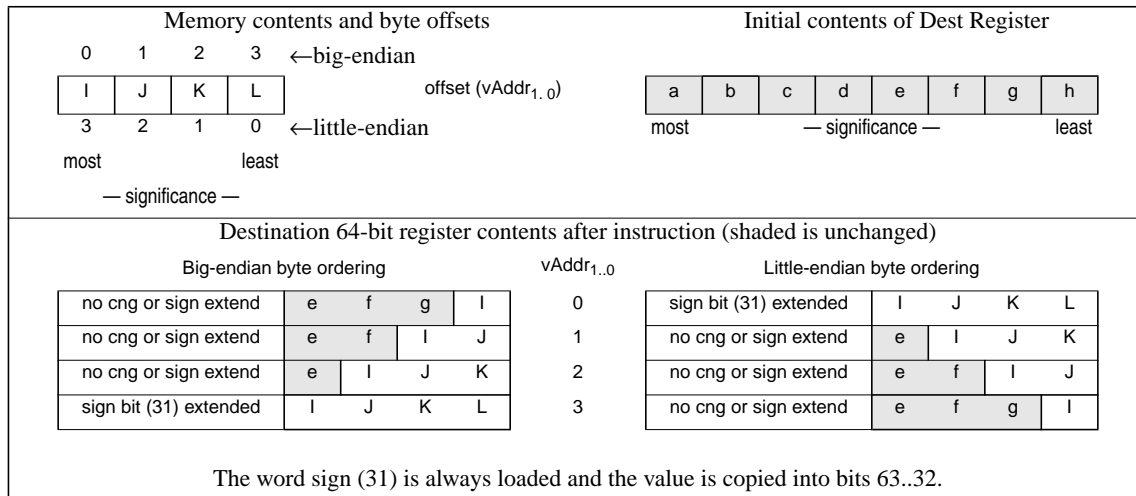
Programming Notes:

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.

Figure 5.21 Bytes Loaded by LWR 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 ← 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

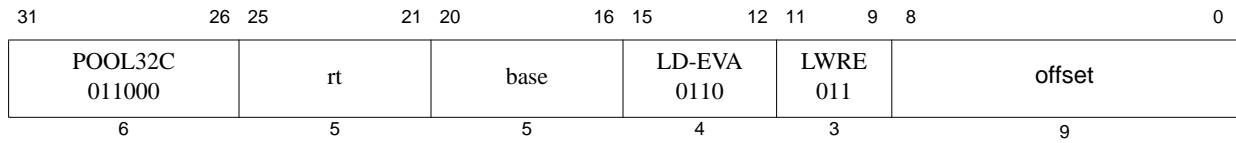
Programming Notes:

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.



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

microMIPS

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

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.

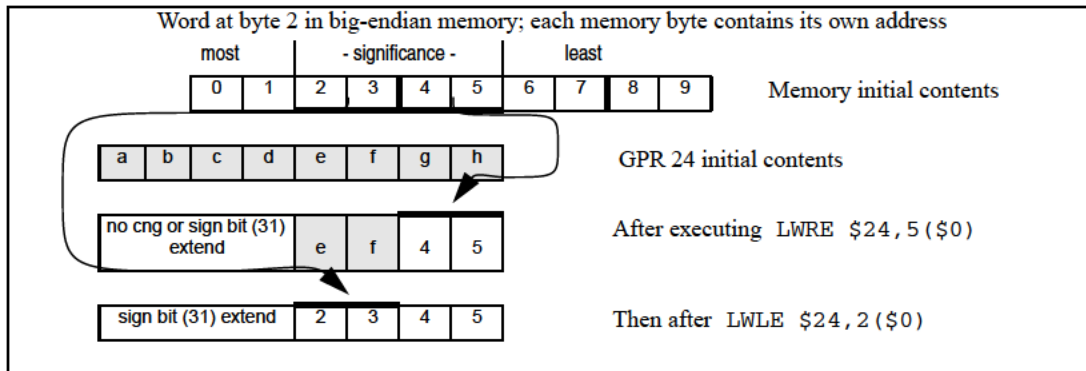
Executing both LWRE and LWLE, in either order, delivers a sign-extended word value in the destination register.

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.

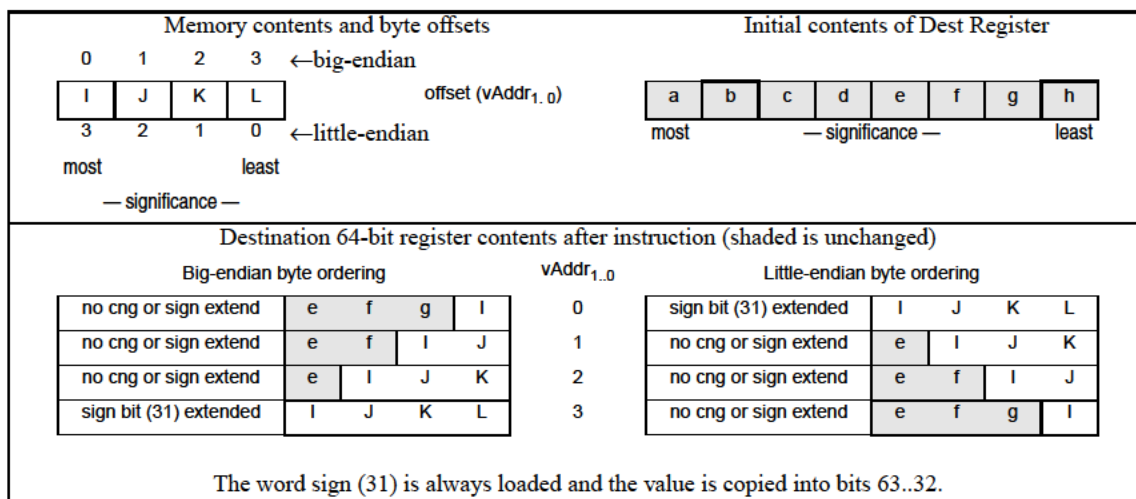
Implementation of this instruction is specified by the *Config5_{EVA}* field being set to one.

Figure 5.22 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 5.23 Bytes Loaded by LWRE Instruction

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

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor ReverseEndian3)
if BigEndianMem = 0 then
    pAddr ← pAddr_PSIZE-1..3 || 03
endif
byte ← vAddr_1..0 xor BigEndianCPU2
word ← vAddr_2 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:

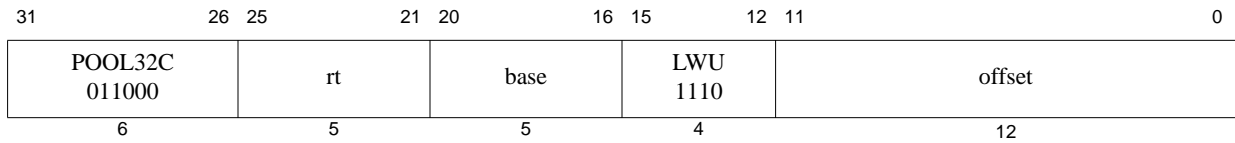
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch, Reserved Instruction, Coprocessor Unusable

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.



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

microMIPS64

Purpose: Load Word Unsigned

To load a word from memory as an unsigned value

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

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 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

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

Operation:

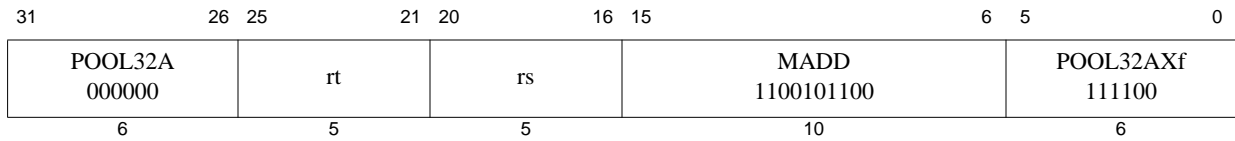
```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
memdoubleword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte ← vAddr2..0 xor (BigEndianCPU || 02)
GPR[rt] ← 032 || memdoubleword31+8*byte..8*byte

```

Exceptions:

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



Format: MADD *rs*, *rt*

microMIPS

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 ← (HI31..0 || LO31..0) + (GPR[rs]31..0 × GPR[rt]31..0)
HI ← sign_extend(temp63..32)
LO ← sign_extend(temp31..0)

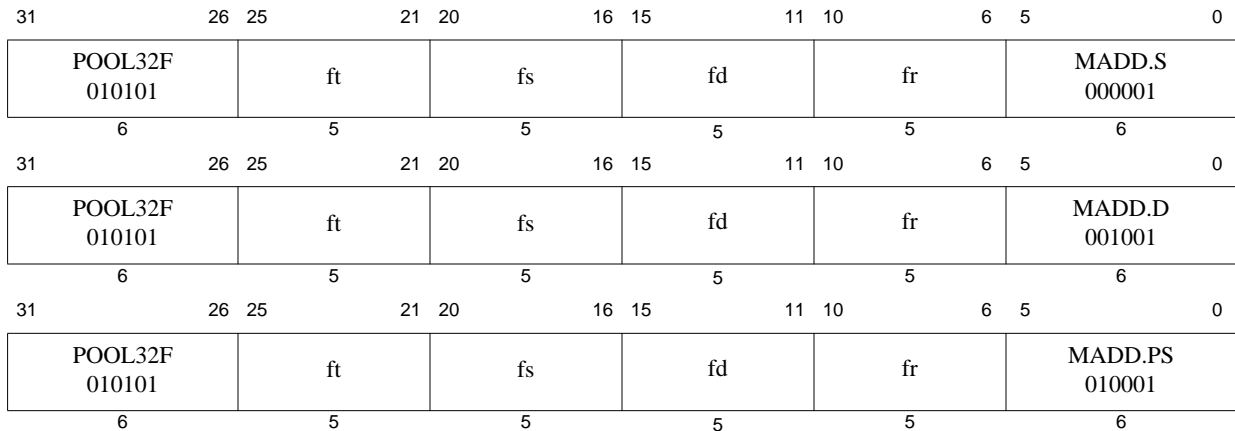
```

Exceptions:

None

Programming Notes:

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



Format: MADD.fmt
MADD.S fd, fr, fs, ft
MADD.D fd, fr, fs, ft
MADD.PS fd, fr, fs, ft

microMIPS
microMIPS
microMIPS

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.

The intermediate product is rounded according to the current rounding mode in *FCSR*. The value in FPR *fr* is added to the product. The result sum is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. The results and flags are as if separate floating-point multiply and add instructions were executed.

MADD.PS multiplies then adds the upper and lower halves of FPR *fr*, FPR *fs*, and FPR *ft* independently, and ORs together any generated exceptional conditions.

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

Restrictions:

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

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

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

Compatibility and Availability:

MADD.S and MADD.D: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode ($FIR_{F64}=0$ or 1, $Status_{FR}=0$ or 1).

Operation:

$vfr \leftarrow \text{ValueFPR}(fr, fmt)$
 $vfs \leftarrow \text{ValueFPR}(fs, fmt)$

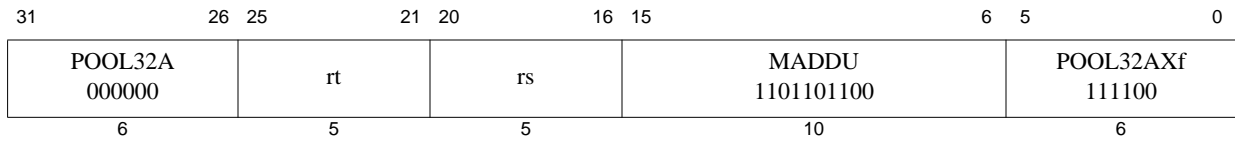
```
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, (vfs ×fmt vft) +fmt vfr)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow



Format: MADDU *rs*, *rt*

microMIPS

Purpose: Multiply and Add Unsigned Word to Hi,Lo

To 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 ← (HI31..0 || LO31..0) + ((032 || GPR[rs]31..0) × (032 || GPR[rt]31..0)
HI ← sign_extend(temp63..32)
LO ← sign_extend(temp31..0)

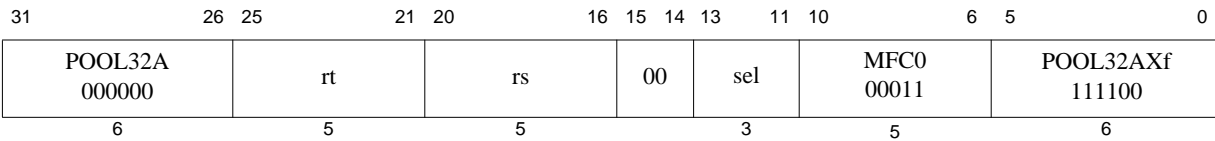
```

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.



Format: MFC0 rt, rs
MFC0 rt, rs, sel

microMIPS
microMIPS

Purpose: Move from Coprocessor 0

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

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

The contents of the coprocessor 0 register specified by the combination of *rs* 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 *rs* and *sel*.

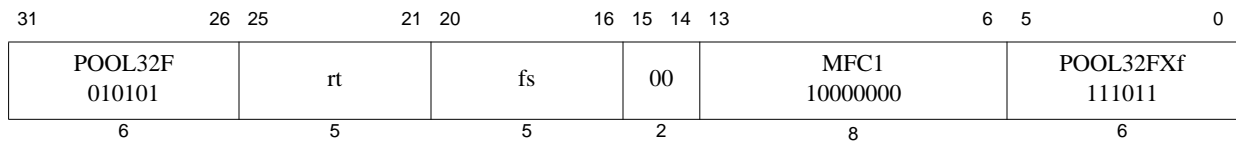
Operation:

```
reg = rs
data ← CPR[0,reg,sel]31..0
if (reg,sel = EntryLo1 or reg,sel = EntryLo0 then
    GPR[rt]29..0 ← data29..0
    GPR[rt]31 ← data63
    GPR[rt]30 ← data62
    GPR[rt]63..32 ← sign_extend(data63)
else
    GPR[rt] ← sign_extend(data)
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: MFC1 rt, fs

microMIPS

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:

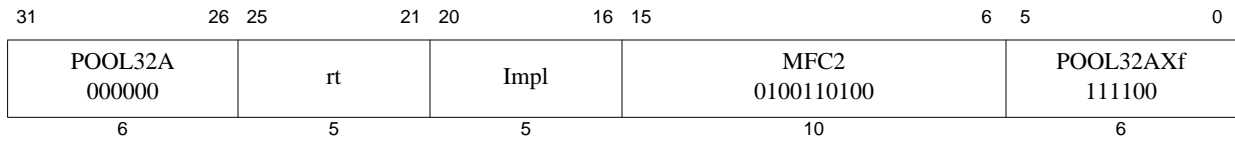
```
data ← ValueFPR(fs, UNINTERPRETED_WORD)31..0
GPR[rt] ← sign_extend(data)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Historical Information:

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



Format: MFC2 rt, Impl

microMIPS

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:

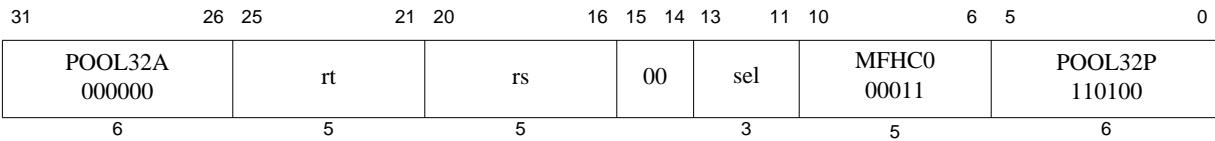
The results are **UNPREDICTABLE** if *impl* specifies a coprocessor 2 register that does not exist.

Operation:

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

Exceptions:

Coprocessor Unusable



Format: MFHC0 *rt*, *rs*
MFHC0 *rt*, *rs*, *sel*

microMIPS Release 5
microMIPS Release 5

Purpose: Move from High Coprocessor 0

To move the contents of the upper 32 bits of a Coprocessor 0 register, extended by 32-bits, to a general register.

Description: $GPR[rt] \leftarrow CPR[0,rs,sel][63:32]$

The contents of the Coprocessor 0 register specified by the combination of *rs* and *sel* are sign-extended and loaded into general register *rt*. Note that not all Coprocessor 0 registers support the *sel* field, and in those instances, the *sel* field must be zero.

When the Coprocessor 0 register specified is the *EntryLo0* or the *EntryLo1* register, MFHC0 must undo the effects of MTHC0, that is, bits 31:30 of the register must be returned as bits 1:0 of the GPR, and bits 32 and those of greater significance must be left-shifted by two and written to bits 31:2 of the GPR.

This feature supports MIPS32 backward-compatibility of MIPS64 systems.

Restrictions:

The results are **UNDEFINED** if Coprocessor 0 does not contain a register as specified by *rs* and *sel*, or the register exists but is not extended by 32-bits, or the register is extended for XPA, but XPA is not supported or enabled.

Operation:

PABITS is the total number of physical address bits implemented. PABITS is defined in the descriptions of *EntryLo0* and *EntryLo1*.

```

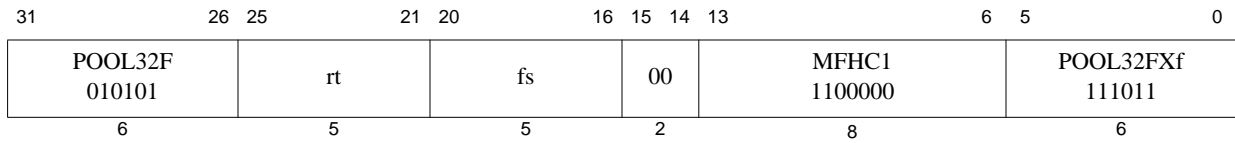
reg ← rs
data ← CPR[0,reg,sel]
if (reg,sel = EntryLo1 or reg,sel = EntryLo0) then
  if (Config3LPA = 1 and PageGrainELPA = 1) then // PABITS > 36
    GPR[rt]31:0 ← data61..30
    GPR[rt]63..32 ← (data61)32 // sign-extend
  endif
else
  GPR[rt] ← sign_extend(data63..32)
endif

```

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: MFHC1 *rt*, *fs*

microMIPS

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 $Status_{FR} = 0$ and *fs* is odd.

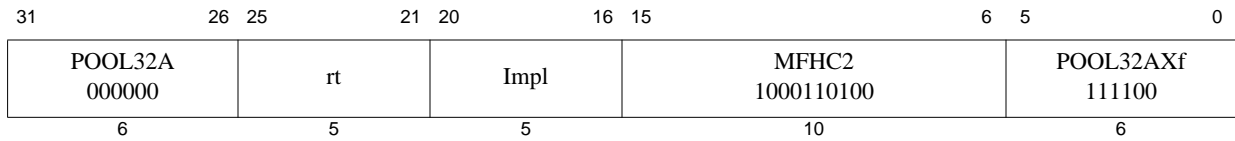
Operation:

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

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: MFHC2 *rt*, *Impl*

microMIPS

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

Purpose: Move Word From High Half of Coprocessor 2 Register

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

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

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

Restrictions:

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

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

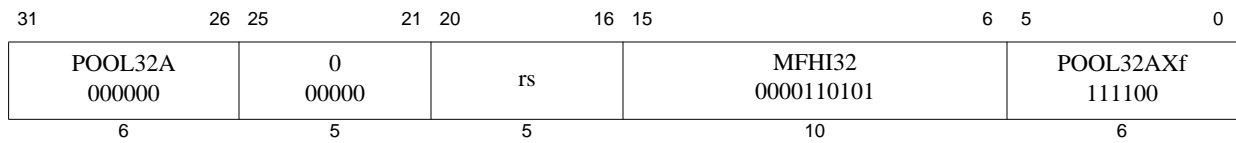
Operation:

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

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: MFHI rs

microMIPS

Purpose: Move From HI Register

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

Description: $GPR[rs] \leftarrow HI$

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

Restrictions:

None

Operation:

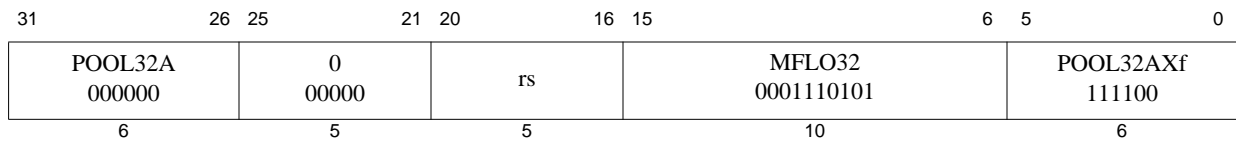
$GPR[rs] \leftarrow HI$

Exceptions:

None

Historical Information:

In the MIPS I, II, and III architectures, the two instructions which follow the MFHI must not modify the *HI* register. If this restriction is violated, the result of the MFHI is **UNPREDICTABLE**. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.



Format: MFLO rs

microMIPS

Purpose: Move From LO Register

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

Description: $GPR[rs] \leftarrow LO$

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

Restrictions:

None

Operation:

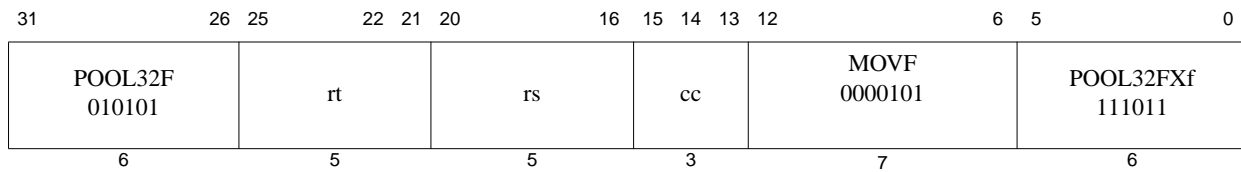
$GPR[rs] \leftarrow LO$

Exceptions:

None

Historical Information:

In the MIPS I, II, and III architectures, the two instructions which follow the MFLO must not modify the *HI* register. If this restriction is violated, the result of the MFLO is **UNPREDICTABLE**. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.



Format: MOVF *rt*, *rs*, *cc*

microMIPS

Purpose: Move Conditional on Floating Point False

To test an FP condition code then conditionally move a GPR

Description: if $\text{FPConditionCode}(cc) = 0$ then $\text{GPR}[rt] \leftarrow \text{GPR}[rs]$

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

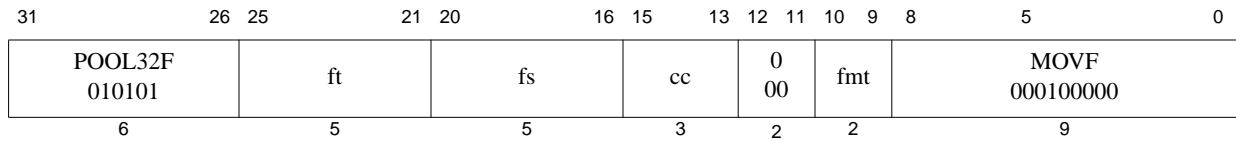
Restrictions:

Operation:

```
if FPConditionCode(cc) = 0 then
    GPR[rt] ← GPR[rs]
endif
```

Exceptions:

Reserved Instruction, Coprocessor Unusable



Format: MOVF.fmt
 MOVF.S ft, fs, cc
 MOVF.D ft, fs, cc
 MOVF.PS ft, fs, cc

microMIPS
 microMIPS
 microMIPS
 microMIPS

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[ft] ← FPR[fs]

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

If the condition code is not zero, then FPR *fs* is not copied and FPR *ft* retains its previous value in format *fmt*. If *ft* 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 *ft* becomes **UNPREDICTABLE**.

MOVF.PS conditionally merges the lower half of FPR *fs* into the lower half of FPR *ft* if condition code *CC* is zero, and independently merges the upper half of FPR *fs* into the upper half of FPR *ft* 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 *ft* 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 the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

```

if fmt ≠ PS
  if FPConditionCode(cc) = 0 then
    StoreFPR(ft, fmt, ValueFPR(fs, fmt))
  else
    StoreFPR(ft, fmt, ValueFPR(ft, 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(ft, PS, ByteMerge(mask, ft, fs))
endif

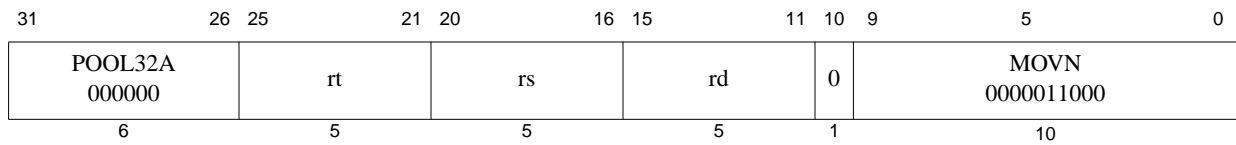
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation



Format: MOVN rd, rs, rt

microMIPS

Purpose: Move Conditional on Not Zero

To conditionally move a GPR after testing a GPR value

Description: if GPR[rt] \neq 0 then GPR[rd] \leftarrow 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:

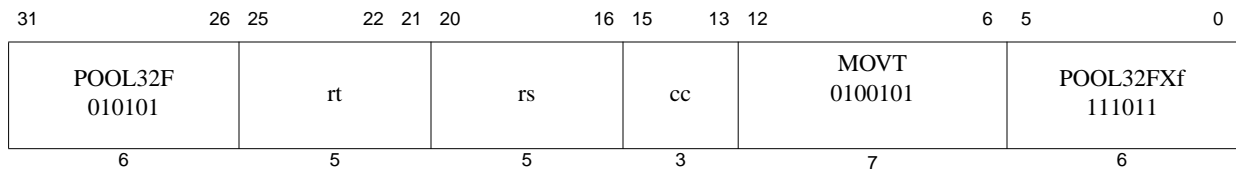
```
if GPR[rt]  $\neq$  0 then
    GPR[rd]  $\leftarrow$  GPR[rs]
endif
```

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.



Format: MOVT *rt*, *rs*, *cc*

microMIPS

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[*rt*] ← GPR[*rs*]

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

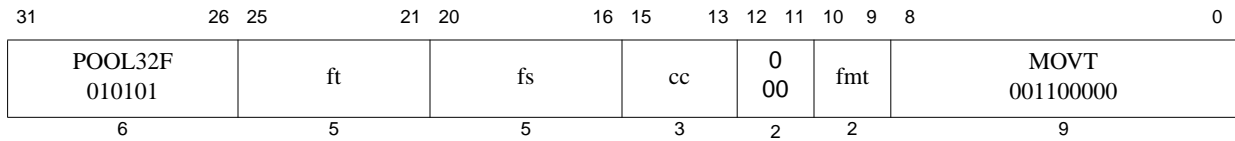
Restrictions:

Operation:

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

Exceptions:

Reserved Instruction, Coprocessor Unusable



Format: MOVT.fmt
 MOVT.S ft, fs, cc
 MOVT.D ft, fs, cc
 MOVT.PS ft, fs, cc

microMIPS
 microMIPS
 microMIPS

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[ft] ← FPR[fs]

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

If the condition code is not one, then FPR *fs* is not copied and FPR *ft* contains its previous value in format *fmt*. If *ft* 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 *ft* becomes **UNPREDICTABLE**.

MOVT.PS conditionally merges the lower half of FPR *fs* into the lower half of FPR *ft* if condition code *CC* is one, and independently merges the upper half of FPR *fs* into the upper half of FPR *ft* if condition code *CC*+1 is one. The *CC* field should be even; if it is odd, the result of this operation is **UNPREDICTABLE**.

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

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**. The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

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

Operation:

```

if fmt ≠ PS
  if FPConditionCode(cc) = 1 then
    StoreFPR(ft, fmt, ValueFPR(fs, fmt))
  else
    StoreFPR(ft, fmt, ValueFPR(ft, 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(ft, PS, ByteMerge(mask, ft, fs))
endif

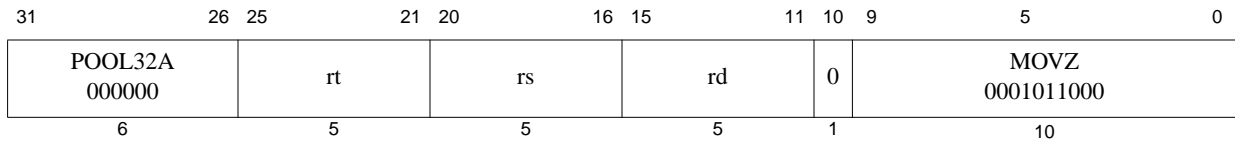
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation



Format: MOVZ rd, rs, rt

microMIPS

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

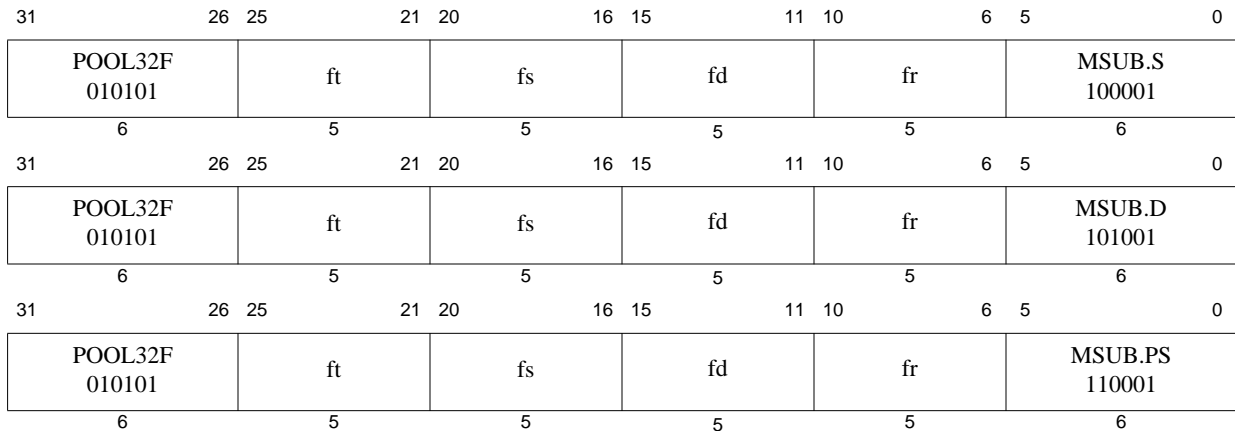
```

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.



Format: MSUB.fmt
 MSUB.S fd, fr, fs, ft
 MSUB.D fd, fr, fs, ft
 MSUB.PS fd, fr, fs, ft

microMIPS
 microMIPS
 microMIPS

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. The intermediate product is rounded according to the current rounding mode in *FCSR*. The subtraction result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. The results and flags are as if separate floating-point multiply and subtract instructions were executed.

MSUB.PS multiplies then subtracts the upper and lower halves of FPR *fr*, FPR *fs*, and FPR *ft* independently, and ORs together any generated exceptional conditions.

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

Restrictions:

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

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

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

Compatibility and Availability:

MSUB.S and MSUB.D: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode ($FIR_{F64}=0$ or 1, $Status_{FR}=0$ or 1).

Operation:

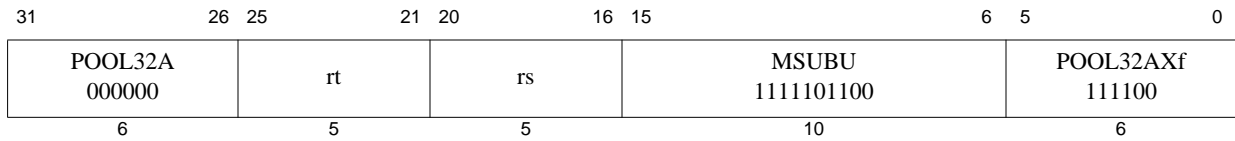
$vfr \leftarrow \text{ValueFPR}(fr, fmt)$
 $vfs \leftarrow \text{ValueFPR}(fs, fmt)$
 $vft \leftarrow \text{ValueFPR}(ft, fmt)$
 $\text{StoreFPR}(fd, fmt, (vfs \times_{fmt} vft) -_{fmt} vfr)$

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow



Format: MSUBU *rs*, *rt*

microMIPS

Purpose: Multiply and Subtract Word to Hi,Lo

To 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)

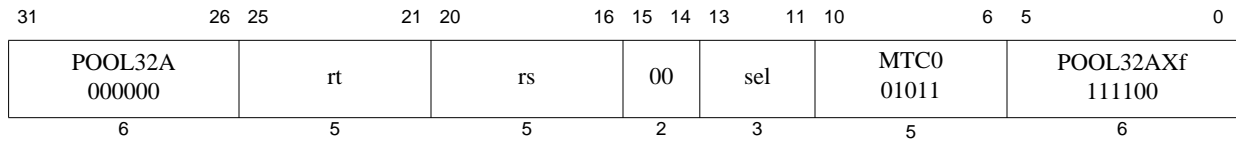
```

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.



Format: MTC0 rt, rs
MTC0 rt, rs, sel

microMIPS
microMIPS

Purpose: Move to Coprocessor 0

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

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

The contents of general register *rt* are loaded into the coprocessor 0 register specified by the combination of *rs* and *sel*. Not all coprocessor 0 registers support the *sel* field. In those instances, the *sel* field must be set to zero.

When the COP0 destination register specified is the *EntryLo0* or the *EntryLo1* register, bits 31:30 appear in the RI/XI fields of the destination register. This feature supports MIPS32 backward compatibility on a MIPS64 system.

Restrictions:

The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *rs* and *sel*.

Operation:

```

data ← GPR[rt]
reg ← rs
if (reg,sel = EntryLo1 or EntryLo0) then
    CPR[0,reg,sel]29..0 ← data29..0
    CPR[0,reg,sel]63 ← data31
    CPR[0,reg,sel]62 ← data30
    CPR[0,reg,sel]61:30 ← 032
else if (Width(CPR[0,reg,sel]) = 64) then
    CPR[0,reg,sel] ← data
else
    CPR[0,reg,sel] ← data31..0

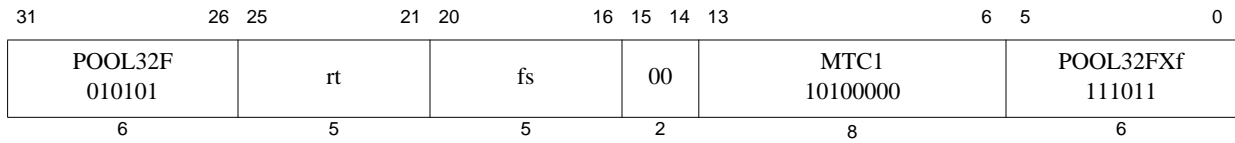
endif

```

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: MTC1 *rt*, *fs*

microMIPS

Purpose: Move Word to Floating Point

To copy a word from a GPR to an FPU (CPI) 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 ← GPR[rt]31..0
StoreFPR(fs, UNINTERPRETED_WORD, data)
```

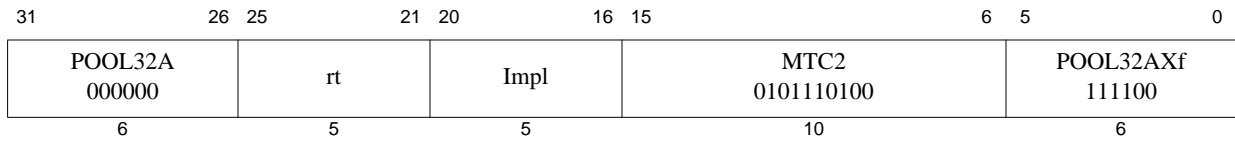
Exceptions:

Coprocessor Unusable

Historical Information:

For MIPS I, MIPS II, and MIPS III the value of FPR *fs* is **UNPREDICTABLE** for the instruction immediately following MTC1.





Format: MTC2 rt, Impl

microMIPS

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 a Coprocessor 2 general register denoted by the *Impl* field. If Coprocessor 2 general registers are 64 bits wide; bits 63..32 of the register denoted by the *Impl* field become **UNPREDICTABLE**. The interpretation of the *Impl* field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:

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

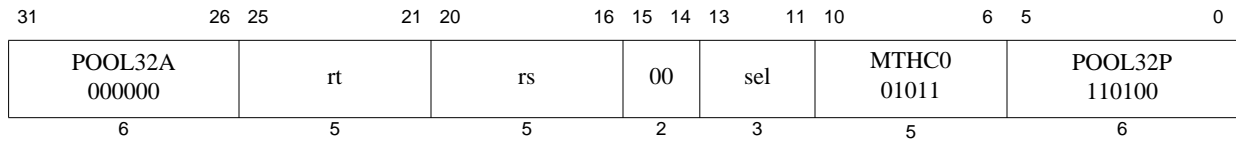
Operation:

$$\begin{aligned} \text{data} &\leftarrow GPR[rt]_{31..0} \\ CP2CPR[Impl] &\leftarrow \text{data} \end{aligned}$$

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: MTHC0 rt, rs
MTHC0 rt, rs, sel

microMIPS Release 5
microMIPS Release 5

Purpose: Move to High Coprocessor 0

To copy a word from a GPR to the upper 32 bits of a COP2 general register that has been extended by 32 bits.

Description: $CPR[0, rs, sel][63:32] \leftarrow GPR[rt]$

The contents of general register *rt* are loaded into the Coprocessor 0 register specified by the combination of *rs* and *sel*. Not all Coprocessor 0 registers support the *sel* field, and when this is the case, the *sel* field must be set to zero.

When the Coprocessor 0 destination register specified is the *EntryLo0* or *EntryLo1* register, bits 1:0 of the GPR appear at bits 31:30 of *EntryLo0* or *EntryLo1*. This is to compensate for *Rl* and *Xl*, which were shifted to bits 63:62 by MTC0 to *EntryLo0* or *EntryLo1*. If *Rl/Xl* are not supported, the shift must still occur, but an MFHC0 instruction will return 0s for these two fields. The GPR is right-shifted by two to vacate the lower two bits, and two 0s are shifted in from the left. The result is written to the upper 32 bits of MIPS64 *EntryLo0* or *EntryLo1*, excluding *Rl/Xl*, which were placed in bits 63:62, i.e., the write must appear atomic, as if both MTC0 and MTHC0 occurred together.

This feature supports MIPS32 backward compatibility of MIPS64 systems.

Restrictions:

The results are **UNDEFINED** if Coprocessor 0 does not contain a register as specified by *rs* and *sel*, or if the register exists but is not extended by 32 bits, or the register is extended for XPA, but XPA is not supported or enabled.

In a 64-bit processor, the MTHC0 instruction writes only the lower 32 bits of register *rt* into the upper 32 bits of the Coprocessor register specified by *rd* and *sel* if that register is extended by MIPS32 Release 5. Specifically, the only registers extended by MIPS32 Release 5 are those required for the XPA feature, and those registers are identical to the same registers in the MIPS64 Architecture, other than *EntryLo0* and *EntryLo1*.

Operation:

```

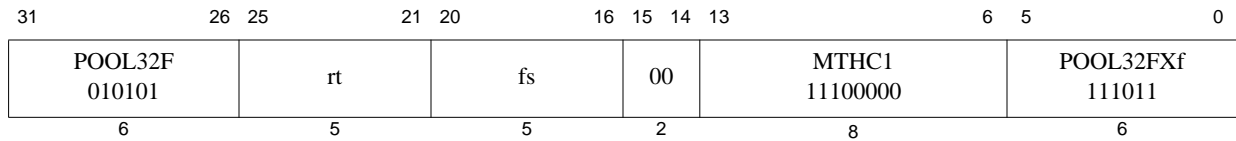
data ← GPR[rt]
reg ← rs
if (reg, sel = EntryLo1 or reg, sel = EntryLo0) then
    if (Config3LPA = 1 and PageGrainELPA = 1) then // PABITS > 36
        CPR[0, reg, sel]31..30 ← data1..0
        CPR[0, reg, sel]61:32 ← data31..2 and ((1<<(PABITS-36))-1)
        CPR[0, reg, sel]63:62 ← 02
    endif
else
    CPR[0, reg, sel][63:32] ← data31..0
endif

```

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: MTHC1 rt, fs

microMIPS

Purpose: Move Word to High Half of Floating Point Register

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

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

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

Restrictions:

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

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

Operation:

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

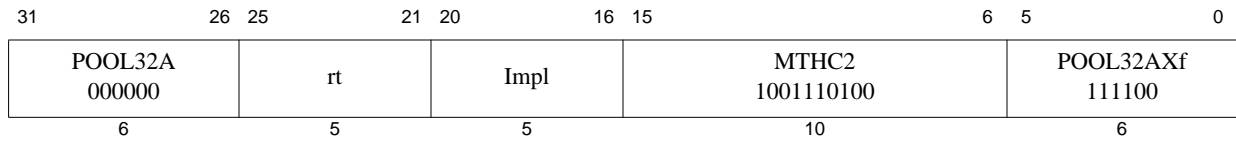
Exceptions:

Coprocessor Unusable

Reserved Instruction

Programming Notes

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



Format: MTHC2 *rt*, *Impl*

microMIPS

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

Purpose: Move Word to High Half of Coprocessor 2 Register

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

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

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

Restrictions:

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

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

Operation:

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

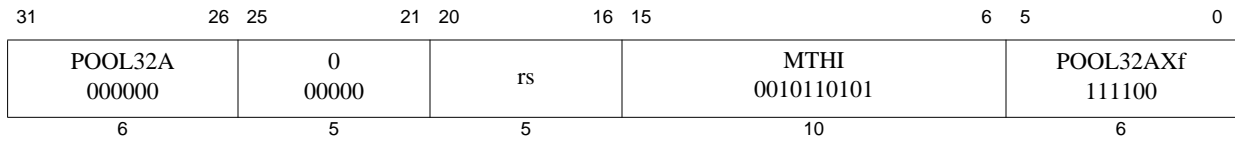
Exceptions:

Coprocessor Unusable

Reserved Instruction

Programming Notes

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



Format: MTHI rs

microMIPS

Purpose: Move to HI Register

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

Description: $HI \leftarrow GPR[rs]$

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:

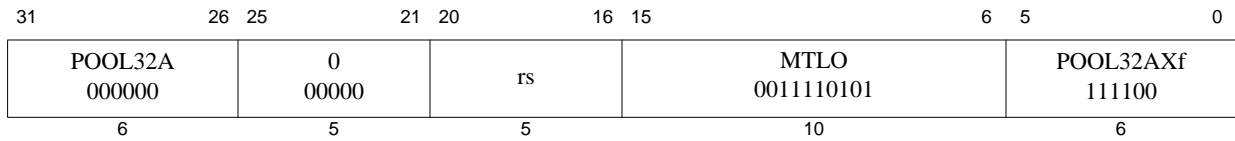
$HI \leftarrow GPR[rs]$

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.



Format: MTLO rs

microMIPS

Purpose: Move to LO Register

To 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:

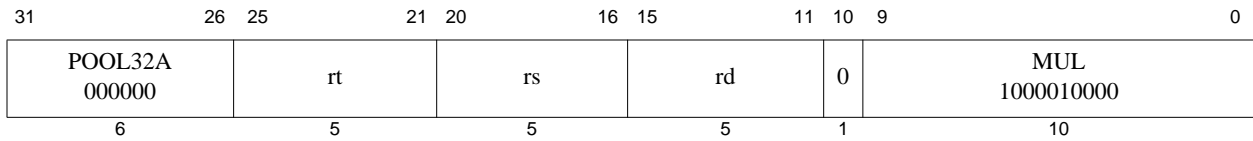
$LO \leftarrow GPR[rs]$

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.



Format: MUL rd, rs, rt

microMIPS

Purpose: Multiply Word to GPR

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

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

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 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:

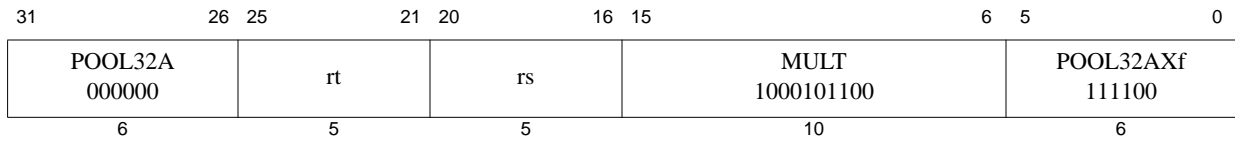
None

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.



Format: MULT *rs*, *rt*

microMIPS

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:

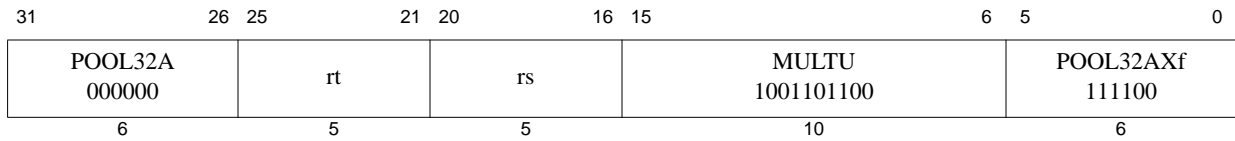
None

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.



Format: MULTU *rs*, *rt*

microMIPS

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:

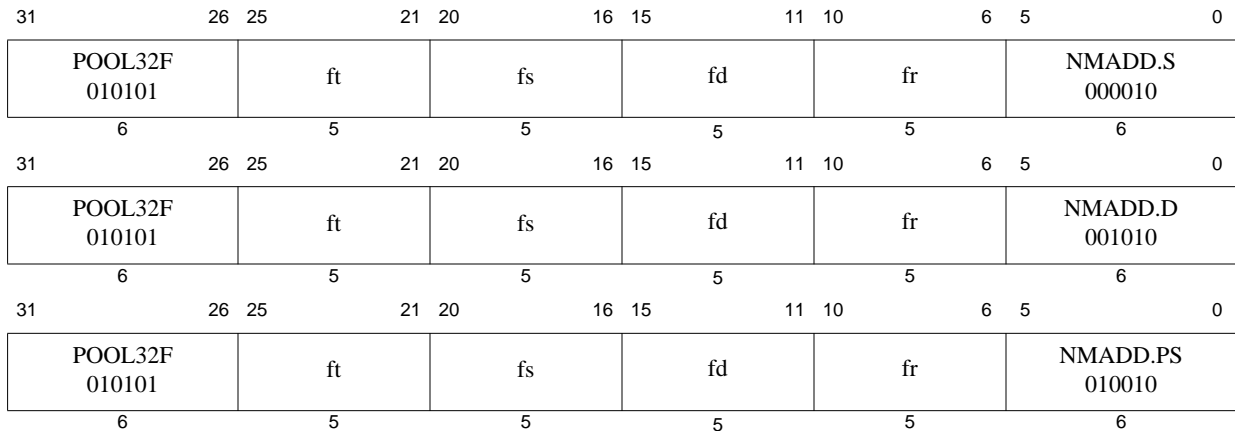
None

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.



Format: NMADD.fmt
 NMADD.S fd, fr, fs, ft
 NMADD.D fd, fr, fs, ft
 NMADD.PS fd, fr, fs, ft

microMIPS
 microMIPS
 microMIPS

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. The intermediate product is rounded according to the current rounding mode in *FCSR*. The value in FPR *fr* is added to the product.

The result sum is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, negated by changing the sign bit, and placed into FPR *fd*. The operands and result are values in format *fmt*. The results and flags are as if separate floating-point multiply and add and negate instructions were executed.

NMADD.PS applies the operation to the upper and lower halves of FPR *fr*, FPR *fs*, and FPR *ft* independently, and ORs together any generated exceptional conditions.

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

Restrictions:

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

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

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

Compatibility and Availability:

NMADD.S and NMADD.D: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode ($FIR_{F64}=0$ or 1, $Status_{FR}=0$ or 1).

Operation:

$$vfr \leftarrow \text{ValueFPR}(fr, fmt)$$

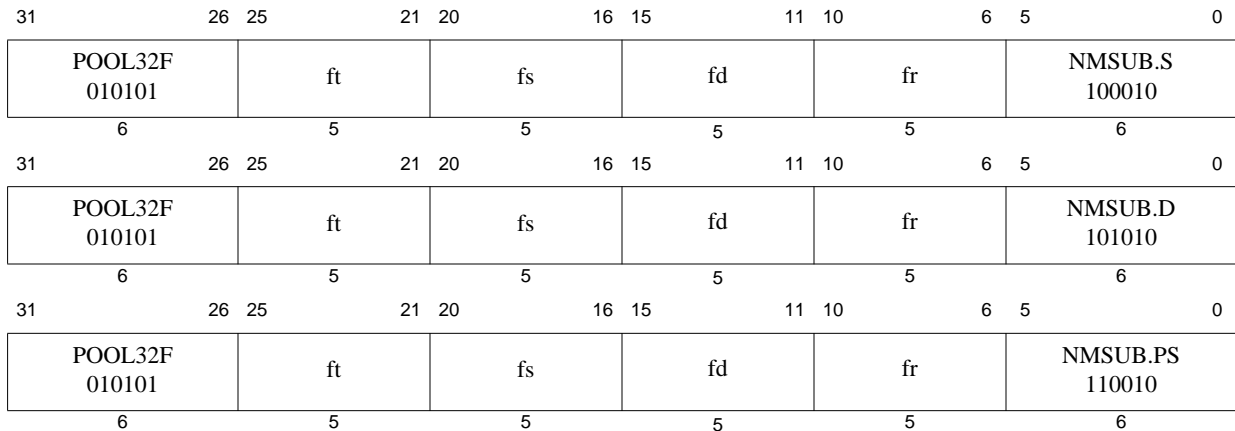
```
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, -(vfr +fmt (vfs ×fmt vft)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow



Format: NMSUB.fmt
 NMSUB.S fd, fr, fs, ft
 NMSUB.D fd, fr, fs, ft
 NMSUB.PS fd, fr, fs, ft

microMIPS
 microMIPS
 microMIPS

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. The intermediate product is rounded according to the current rounding mode in *FCSR*. The value in FPR *fr* is subtracted from the product.

The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, negated by changing the sign bit, and placed into FPR *fd*. The operands and result are values in format *fmt*. The results and flags are as if separate floating-point multiply and subtract and negate instructions were executed.

NMSUB.PS applies the operation to the upper and lower halves of FPR *fr*, FPR *fs*, and FPR *ft* independently, and ORs together any generated exceptional conditions.

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

Restrictions:

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

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

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

Compatibility and Availability:

NMSUB.S and NMSUB.D: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode ($FIR_{F64}=0$ or 1, $Status_{FR}=0$ or 1).

Operation:

$vfr \leftarrow \text{ValueFPR}(fr, fmt)$

```
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	5	0
POOL32A 000000	0 00000	0 00000	0 00000	0 00000	0 00000	SLL 000000
6	5	5	5	5	5	6

Format: NOP

Assembly Idiom microMIPS

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:

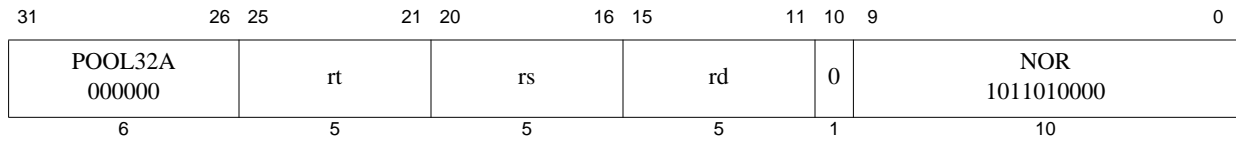
None

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.



Format: NOR *rd*, *rs*, *rt*

microMIPS

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:

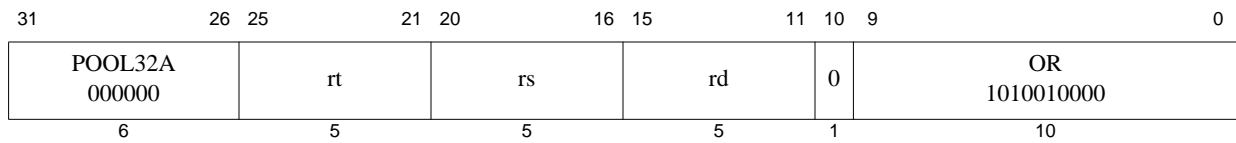
None

Operation:

$GPR[rd] \leftarrow GPR[rs] \text{ nor } GPR[rt]$

Exceptions:

None



Format: OR *rd*, *rs*, *rt*

microMIPS

Purpose: Or

To do a bitwise logical OR

Description: $GPR[rd] \leftarrow GPR[rs] \text{ or } GPR[rt]$

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:

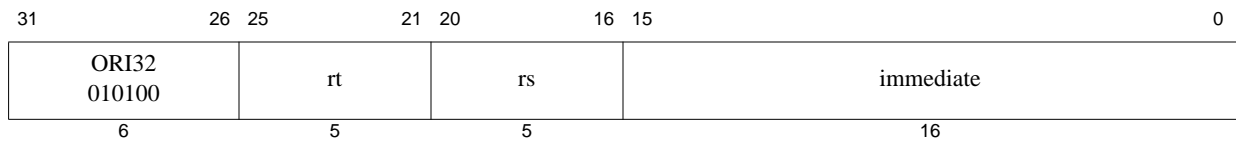
None

Operation:

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

Exceptions:

None



Format: ORI *rt*, *rs*, *immediate*

microMIPS

Purpose: Or Immediate

To do a bitwise logical OR with a constant

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

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

Restrictions:

None

Operation:

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

Exceptions:

None

31	26	25			6	5	0
POOL32A 000000	0 00000	0 00000	5 00101	0 00000	SLL 000000		
6	5	5	5	5	6		

Format: PAUSE

microMIPS

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                /* Resume at the following instruction */
    DescheduleInstructionStream()
endif

```

Exceptions:

None

Programming Notes:

The PAUSE instruction is intended to be inserted into the instruction stream after an LL instruction has set the LLBit and found the software lock set. The program may wait forever if a PAUSE instruction is executed and there is no possibility that the LLBit will ever be cleared.

An example use of the PAUSE instruction is included in the following example:

```

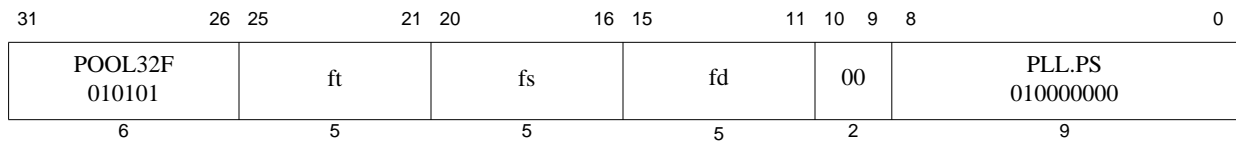
acquire_lock:

```

```
ll    t0, 0(a0)           /* Read software lock, set hardware lock */
bnez  t0, acquire_lock_retry: /* Branch if software lock is taken */
addiu t0, t0, 1           /* Set the software lock */
sc    t0, 0(a0)           /* Try to store the software lock */
bnez  t0, 10f             /* Branch if lock acquired successfully */
sync
acquire_lock_retry:
pause                               /* Wait for LLBIT to clear before retry */
b     acquire_lock             /* and retry the operation */
nop
10:

    Critical region code

release_lock:
sync
sw    zero, 0(a0)         /* Release software lock, clearing LLBIT */
                                /* for any PAUSEd waiters */
```



Format: PLL.PS fd, fs, ft

microMIPS

Purpose: Pair Lower Lower

To merge a pair of paired single values with realignment

Description: $FPR[fd] \leftarrow \text{lower}(FPR[fs]) \ || \ \text{lower}(FPR[ft])$

A new paired-single value is formed by concatenating the lower single of FPR *fs* (bits **31..0**) and the lower single of FPR *ft* (bits **31..0**).

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

Restrictions:

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

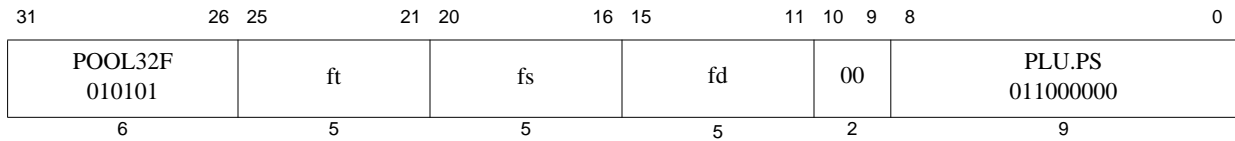
The result of this instruction is **UNPREDICTABLE** if the processor is executing in the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

$\text{StoreFPR}(fd, PS, \text{ValueFPR}(fs, PS)_{31..0} \ || \ \text{ValueFPR}(ft, PS)_{31..0})$

Exceptions:

Coprocessor Unusable, Reserved Instruction



Format: PLU.PS *fd*, *fs*, *ft*

microMIPS

Purpose: Pair Lower Upper

To merge a pair of paired single values with realignment

Description: $\text{FPR}[fd] \leftarrow \text{lower}(\text{FPR}[fs]) \ || \ \text{upper}(\text{FPR}[ft])$

A new paired-single value is formed by concatenating the lower single of FPR *fs* (bits **31..0**) and the upper single of FPR *ft* (bits **63..32**).

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

Restrictions:

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

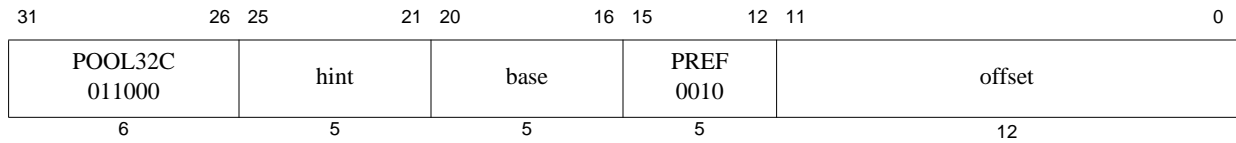
The result of this instruction is **UNPREDICTABLE** if the processor is executing in the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

$\text{StoreFPR}(fd, PS, \text{ValueFPR}(fs, PS)_{31..0} \ || \ \text{ValueFPR}(ft, PS)_{63..32})$

Exceptions:

Coprocessor Unusable, Reserved Instruction



Format: `PREF hint,offset(base)`

microMIPS

Purpose: Prefetch

To move data between memory and cache.

Description: `prefetch_memory(GPR[base] + offset)`

PREF adds the 12-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 6.25 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 6.25 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.
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:

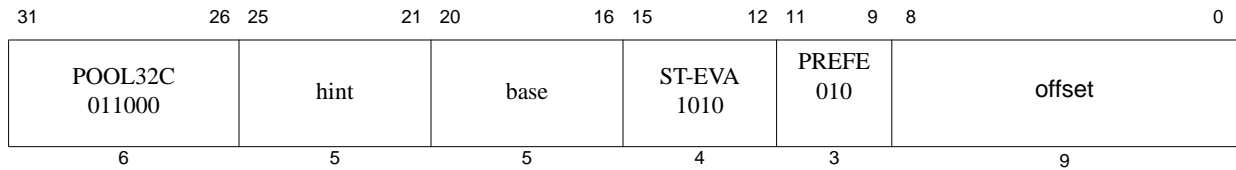
Prefetch cannot move data to or from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

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.



Format: `PREFE hint,offset(base)`

microMIPS

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 6.26 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 6.26 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:

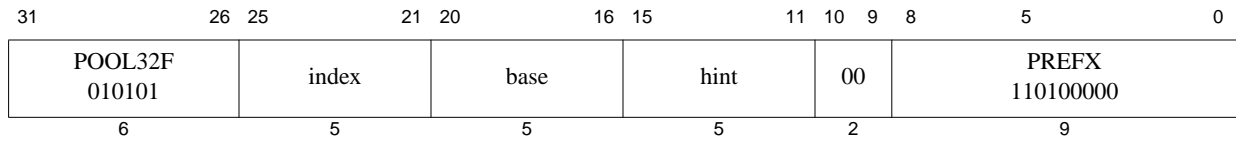
Prefetch cannot move data to or from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

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.



Format: `PREFX hint, index(base)`

microMIPS
microMIPS

Purpose: Prefetch Indexed

To move data between memory and cache.

Description: `prefetch_memory[GPR[base] + GPR[index]]`

PREFX adds the contents of GPR *index* to the contents of GPR *base* to form an effective byte address. The *hint* field supplies information about the way the data is expected to be used.

The only functional difference between the PREF and PREFX instructions is the addressing mode implemented by the two. Refer to the [PREF](#) instruction for all other details, including the encoding of the *hint* field.

Restrictions:

Compatibility and Availability:

PREFX: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode ($FIR_{F64}=0$ or 1, $Status_{FR}=0$ or 1).

Operation:

```
vAddr ← GPR[base] + GPR[index]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

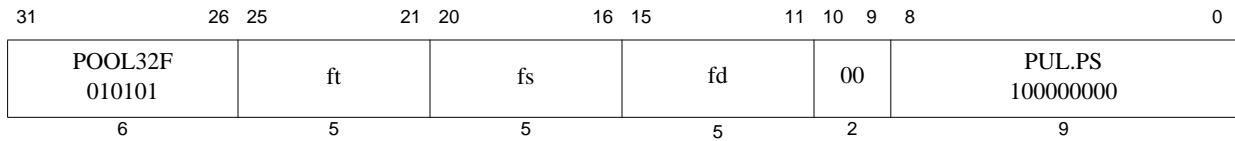
Exceptions:

Coprocessor Unusable, Reserved Instruction, Bus Error, Cache Error

Programming Notes:

The PREFX instruction is only available on processors that implement floating point and should never be generated by compilers in situations other than those in which the corresponding load and store indexed floating point instructions are generated.

Refer to the corresponding section in the [PREF](#) instruction description.



Format: PUL.PS *fd*, *fs*, *ft*

microMIPS

Purpose: Pair Upper Lower

To merge a pair of paired single values with realignment

Description: $\text{FPR}[fd] \leftarrow \text{upper}(\text{FPR}[fs]) \mid \mid \text{lower}(\text{FPR}[ft])$

A new paired-single value is formed by concatenating the upper single of FPR *fs* (bits **63..32**) and the lower single of FPR *ft* (bits **31..0**).

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

Restrictions:

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

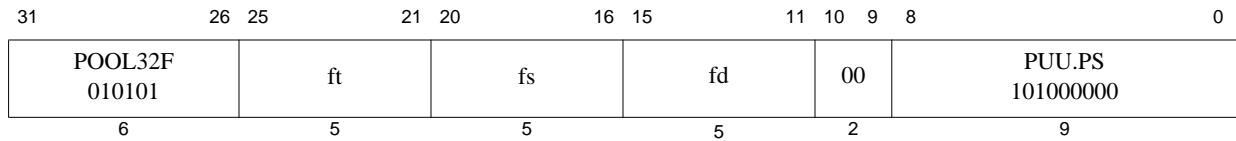
The result of this instruction is **UNPREDICTABLE** if the processor is executing in the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

$\text{StoreFPR}(fd, PS, \text{ValueFPR}(fs, PS)_{63..32} \mid \mid \text{ValueFPR}(ft, PS)_{31..0})$

Exceptions:

Coprocessor Unusable, Reserved Instruction



Format: PUU.PS *fd*, *fs*, *ft*

microMIPS

Purpose: Pair Upper Upper

To merge a pair of paired single values with realignment

Description: $FPR[fd] \leftarrow upper(FPR[fs]) \ || \ upper(FPR[ft])$

A new paired-single value is formed by concatenating the upper single of FPR *fs* (bits **63..32**) and the upper single of FPR *ft* (bits **63..32**).

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

Restrictions:

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

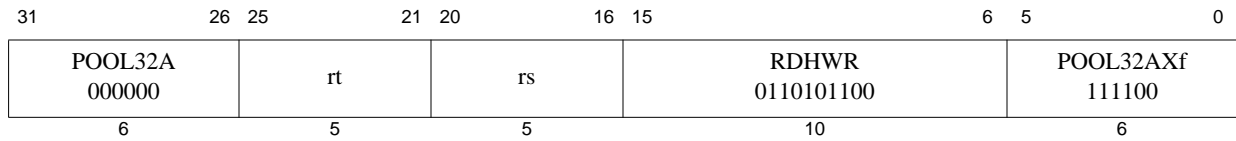
The result of this instruction is **UNPREDICTABLE** if the processor is executing in the *FR*=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR*=1 mode, but not with *FR*=0, and not on a 32-bit FPU.

Operation:

$StoreFPR(fd, PS, ValueFPR(fs, PS)_{63..32} \ || \ ValueFPR(ft, PS)_{63..32})$

Exceptions:

Coprocessor Unusable, Reserved Instruction



Format: RDHWR *rt,rs*

microMIPS

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: $GPR[rt] \leftarrow HWR[rs]$

If access is allowed to the specified hardware register, the contents of the register specified by *rs* 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 *rs* field for each, are shown in [Table 6.27](#).

Table 6.27 RDHWR Register Numbers

Register Number (<i>rd</i> 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</i> _{CPUNum} field.										
1	SYNCL_Step	Address step size to be used with the SYNCL 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: auto; margin-right: auto;"> <thead> <tr> <th>CCRes Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">1</td> <td>CC register increments every CPU cycle</td> </tr> <tr> <td style="text-align: center;">2</td> <td>CC register increments every second CPU cycle</td> </tr> <tr> <td style="text-align: center;">3</td> <td>CC register increments every third CPU cycle</td> </tr> <tr> <td colspan="2" style="text-align: center;">etc.</td> </tr> </tbody> </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 6.27 RDHWR Register Numbers

Register Number (rd Value)	Mnemonic	Description
30-31		These register numbers are reserved for implementation-dependent use. If they are not implemented, access results in a Reserved Instruction Exception.

Restrictions:

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

Access to the specified hardware register is enabled if Coprocessor 0 is enabled, or if the corresponding bit is set in the *HWREna* register. If access is not allowed or the register is not implemented, a Reserved Instruction Exception is signaled.

Operation:

```

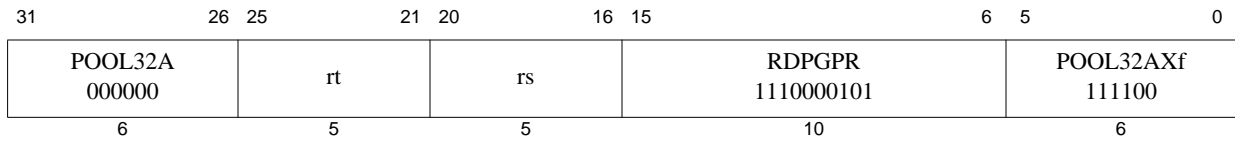
case rs
  0: temp ← sign_extend(EBaseCPUNum)
  1: temp ← sign_extend(SYNCI_StepSize())
  2: temp ← sign_extend(Count)
  3: temp ← sign_extend(CountResolution())
  29: temp ← sign_extend_if_32bit_op(UserLocal)
  30: temp ← sign_extend_if_32bit_op(Implementation-Dependent-Value)
  31: temp ← sign_extend_if_32bit_op(Implementation-Dependent-Value)
  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



Format: RDPGPR *rt*, *rs*

microMIPS

Purpose: Read GPR from Previous Shadow Set

To move the contents of a GPR from the previous shadow set to a current GPR.

Description: $GPR[rt] \leftarrow SGPR[SRSCtl_{PSS}, rs]$

The contents of the shadow GPR register specified by $SRSCtl_{PSS}$ (signifying the previous shadow set number) and *rs* (specifying the register number within that set) is moved to the current GPR *rt*.

Restrictions:

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

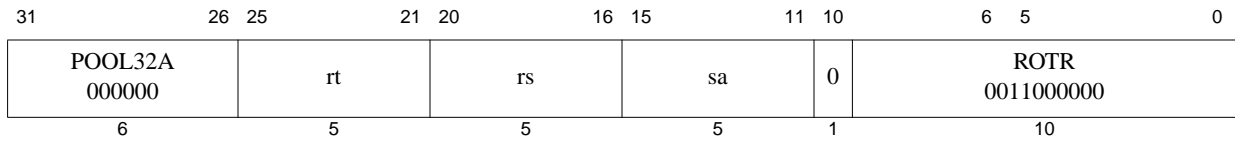
Operation:

$GPR[rt] \leftarrow SGPR[SRSCtl_{PSS}, rs]$

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: ROTR *rt*, *rs*, *sa*

SmartMIPS Crypto, microMIPS

Purpose: Rotate Word Right

To execute a logical right-rotate of a word by a fixed number of bits

Description: $GPR[rt] \leftarrow GPR[rs] \leftrightarrow(\text{right}) sa$

The contents of the low-order 32-bit word of GPR *rs* are rotated right; the word result is sign-extended and placed in GPR *rt*. 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:

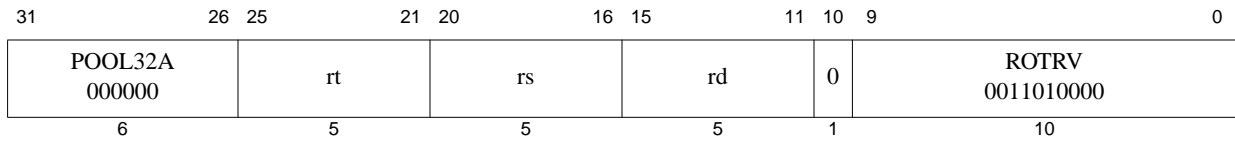
```

if NotWordValue(GPR[rt]) or
  ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
  UNPREDICTABLE
endif
s ← sa
temp ← GPR[rs]s-1..0 || GPR[rs]31..s
GPR[rt] ← sign_extend(temp)

```

Exceptions:

Reserved Instruction



Format: ROTRV rd, rt, rs

SmartMIPS Crypto, microMIPS

Purpose: Rotate Word Right Variable

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

Description: $GPR[rd] \leftarrow GPR[rt] \leftrightarrow(\text{right}) GPR[rs]$

The contents of the low-order 32-bit word of GPR *rt* are rotated right; the word result is sign-extended and placed in GPR *rd*. The bit-rotate amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

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

Operation:

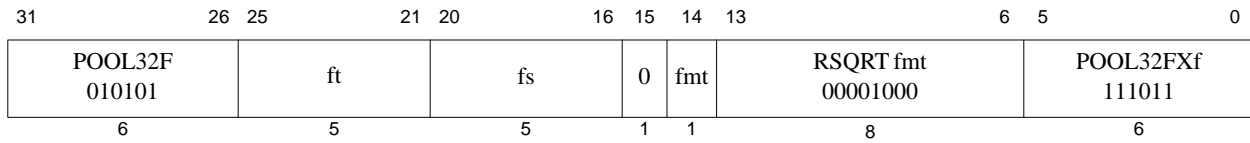
```

if NotWordValue(GPR[rt]) or
  ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
  UNPREDICTABLE
endif
s ← GPR[rs]4..0
temp ← GPR[rt]s-1..0 || GPR[rt]31..s
GPR[rd] ← sign_extend(temp)

```

Exceptions:

Reserved Instruction



Format: RSQRT.fmt
 RSQRT.S ft, fs
 RSQRT.D ft, fs

microMIPS
 microMIPS

Purpose: Reciprocal Square Root Approximation

To approximate the reciprocal of the square root of an FP value (quickly)

Description: $FPR[ft] \leftarrow 1.0 / \text{sqrt}(FPR[fs])$

The reciprocal of the positive square root of the value in FPR *fs* is approximated and placed into FPR *ft*. 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 *ft* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

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

Compatibility and Availability:

RSQRT.S and RSQRT.D: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode ($FIR_{F64}=0$ or 1, $Status_{FR}=0$ or 1).

Operation:

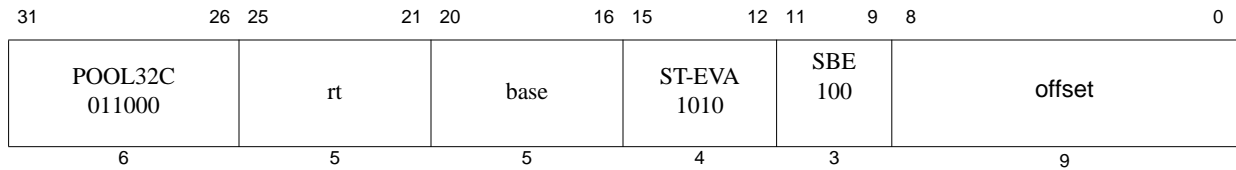
`StoreFPR(ft, fmt, 1.0 / SquareRoot(valueFPR(fs, fmt)))`

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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



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

microMIPS

Purpose: Store Byte EVA

To store a byte to user mode virtual address space when executing in kernel mode.

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

The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

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

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

Restrictions:

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

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← 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

Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable

- Any prefetch that is executed on the processor executing the LL and SC sequence (due to a cache eviction between the LL and SC).
- A non-coherent store executed between an LL and SC sequence to the block of synchronizable physical memory containing the word.
- The instructions executed starting with the LL and ending with the SC do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

CACHE operations that are local to the processor executing the LL/SC sequence will result in unpredictable behaviour of the SC if executed between the LL and SC, that is, they may cause the SC to fail where it could have succeeded. Non-local CACHE operations (address-type with coherent CCA) may cause an SC to fail on either the local processor or on the remote processor in multiprocessor or multi-threaded systems. This definition of the effects of CACHE operations is mandated if *Config5_{LLB}*=1. If *Config5_{LLB}*=0, then CACHE effects are implementation-dependent.

The following conditions must be true or the result of the SC is not predictable—the SC may fail or succeed (if *Config5_{LLB}*=1, then either success or failure is mandated, else the result is **UNPREDICTABLE**):

- Execution of SC must have been preceded by execution of an LL instruction.
- An RMW sequence executed without intervening events that would cause the SC to fail must use the same address in the LL and SC. The address is the *same* if the virtual address, physical address, and cacheability & coherency attribute are identical.

Atomic RMW is provided only for synchronizable memory locations. A synchronizable memory location is one that is associated with the state and logic necessary to implement the LL/SC semantics. Whether a memory location is synchronizable depends on the processor and system configurations, and on the memory access type used for the location:

- **Uniprocessor atomicity:** To provide atomic RMW on a single processor, all accesses to the location must be made with memory access type of either *cached noncoherent* or *cached coherent*. All accesses must be to one or the other access type, and they may not be mixed.
- **MP atomicity:** To provide atomic RMW among multiple processors, all accesses to the location must be made with a memory access type of *cached coherent*.
- **I/O System:** To provide atomic RMW with a coherent I/O system, all accesses to the location must be made with a memory access type of *cached coherent*. If the I/O system does not use coherent memory operations, then atomic RMW cannot be provided with respect to the I/O reads and writes.

Restrictions:

The addressed location must have a memory access type of *cached noncoherent* or *cached coherent*; if it does not, the result is **UNPREDICTABLE**.

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

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
LLbit ← 0 // if Config5LLB=1, SC always clears LLbit regardless of address match.

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

Programming Notes:

LL and SC are used to atomically update memory locations, as shown below.

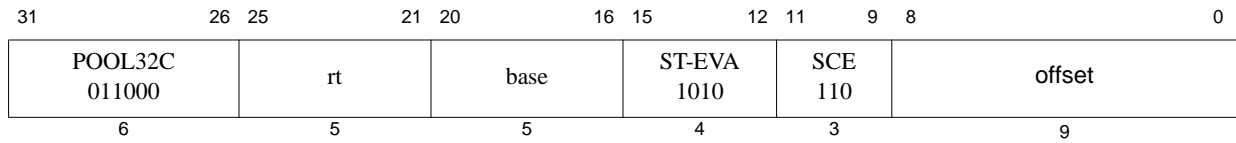
```

L1:
    LL      T1, (T0) # load counter
    ADDI   T2, T1, 1 # increment
    SC     T2, (T0) # try to store, checking for atomicity
    BEQ   T2, 0, L1 # if not atomic (0), try again
    NOP                    # branch-delay slot

```

Exceptions between the LL and SC cause SC to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LL and SC function on a single processor for *cached noncoherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.



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

microMIPS

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 $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{GPR}[\text{rt}]$, $\text{GPR}[\text{rt}] \leftarrow 1$
else $\text{GPR}[\text{rt}] \leftarrow 0$

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 to memory at the location specified by the aligned effective address.
- A 1, indicating success, is written into GPR *rt*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *rt*.

If either of the following events occurs between the execution of LL and SC, the SC fails:

- A coherent store is completed by another processor or coherent I/O module into the block of synchronizable physical memory containing the word. The size and alignment of the block is implementation dependent, but it is at least one word and at most the minimum page size.
- An ERET instruction is executed.

If either of the following events occurs between the execution of LLE and SCE, the SCE may succeed or it may fail; the success or failure is not predictable. Portable programs should not cause one of these events.

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LLE/SCE.
- The instructions executed starting with the LLE and ending with the SCE do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

The following conditions must be true or the result of the SCE is **UNPREDICTABLE**:

- Execution of SCE must have been preceded by execution of an LLE instruction.
- An RMW sequence executed without intervening events that would cause the SCE to fail must use the same address in the LLE and SCE. The address is the same if the virtual address, physical address, and cacheability & coherency attribute are identical.

Atomic RMW is provided only for synchronizable memory locations. A synchronizable memory location is one that

is associated with the state and logic necessary to implement the LLE/SCE semantics. Whether a memory location is synchronizable depends on the processor and system configurations, and on the memory access type used for the location:

- **Uniprocessor atomicity:** To provide atomic RMW on a single processor, all accesses to the location must be made with memory access type of either *cached noncoherent* or *cached coherent*. All accesses must be to one or the other access type, and they may not be mixed.
- **MP atomicity:** To provide atomic RMW among multiple processors, all accesses to the location must be made with a memory access type of *cached coherent*.
- **I/O System:** To provide atomic RMW with a coherent I/O system, all accesses to the location must be made with a memory access type of *cached coherent*. If the I/O system does not use coherent memory operations, then atomic RMW cannot be provided with respect to the I/O reads and writes.

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

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

Restrictions:

The addressed location must have a memory access type of *cached noncoherent* or *cached coherent*; if it does not, the result is **UNPREDICTABLE**.

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

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:

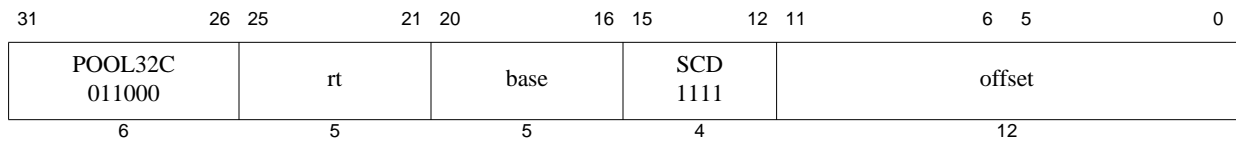
LLE and SCE are used to atomically update memory locations, as shown below.

```
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.



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

microMIPS

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

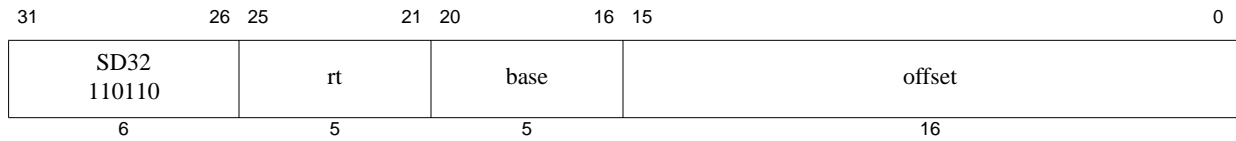
Programming Notes:

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    # branch-delay slot
```

Exceptions between the LLD and SCD cause SCD to fail, so persistent exceptions must be avoided. Some examples of such exceptions are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LLD and SCD function on a single processor for *cached noncoherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.



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

microMIPS64

Purpose: Store Doubleword

To store a doubleword to memory

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

The 64-bit doubleword in GPR *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

Operation:

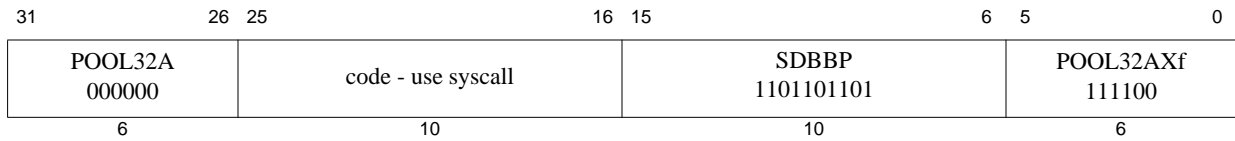
```

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

```

Exceptions:

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



Format: SDBBP code

EJTAG microMIPS

Purpose: Software Debug Breakpoint

To cause a debug breakpoint exception

Description:

This instruction causes a debug exception, passing control to the debug exception handler. If the processor is executing in Debug Mode when the SDBBP instruction is executed, the exception is a Debug Mode Exception, which sets the Debug_{DExcCode} field to the value 0x9 (Bp). The code field can be used for passing information to the debug exception handler, and is retrieved by the debug exception handler only by loading the contents of the memory word containing the instruction, using the DEPC register. The CODE field is not used in any way by the hardware.

Restrictions:

Operation:

```

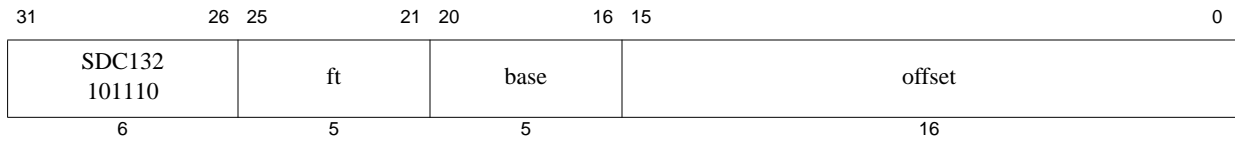
If DebugDM = 0 then
    SignalDebugBreakpointException()
else
    SignalDebugModeBreakpointException()
endif

```

Exceptions:

Debug Breakpoint Exception

Debug Mode Breakpoint Exception



Format: SDC1 ft, offset(base)

microMIPS

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 $\text{EffectiveAddress}_{2..0} \neq 0$ (not doubleword-aligned).

Operation:

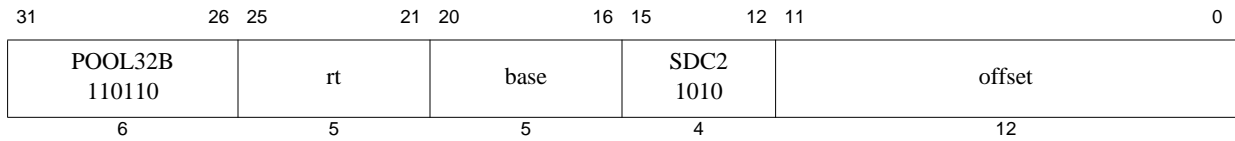
```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← 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



Format: SDC2 rt, offset(base)

microMIPS

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 12-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:

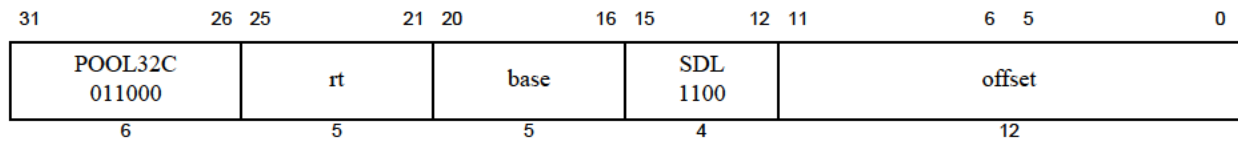
```

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



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

microMIPS64

Purpose: Store Doubleword Left

To store the most-significant part of a doubleword to an unaligned memory address

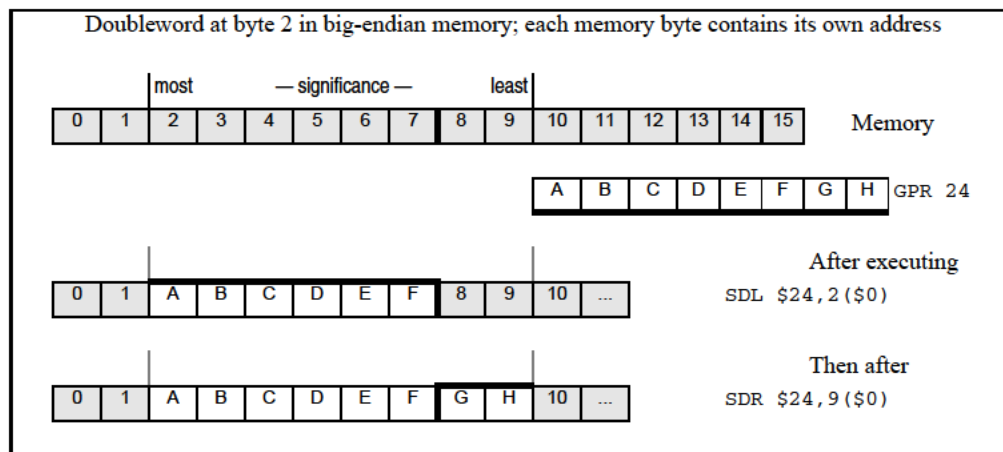
Description: $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{Some_Bytes_From GPR}[\text{rt}]$

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 8 consecutive bytes forming a doubleword (*DW*) in memory, starting at an arbitrary byte boundary.

A part of *DW*, the most-significant 1 to 8 bytes, is in the aligned doubleword containing *EffAddr*. The same number of most-significant (left) bytes of GPR *rt* are stored into these bytes of *DW*.

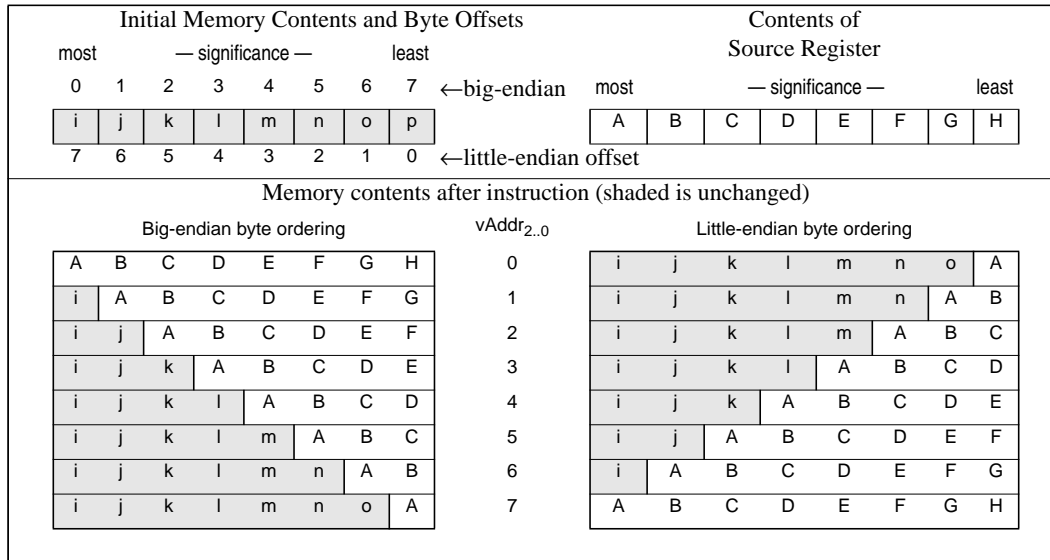
The figure below illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 6 bytes, is located in the aligned doubleword containing the most-significant byte at 2. First, SDL stores the 6 most-significant bytes of the source register into these bytes in memory. Next, the complementary SDR instruction stores the remainder of *DW*.

Figure 6.24 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 6.25 Bytes Stored by an SDL Instruction

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

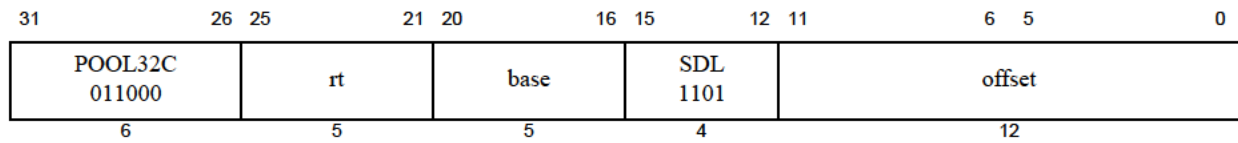
```

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



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

microMIPS64

Purpose: Store Doubleword Right

To store the least-significant part of a doubleword to an unaligned memory address

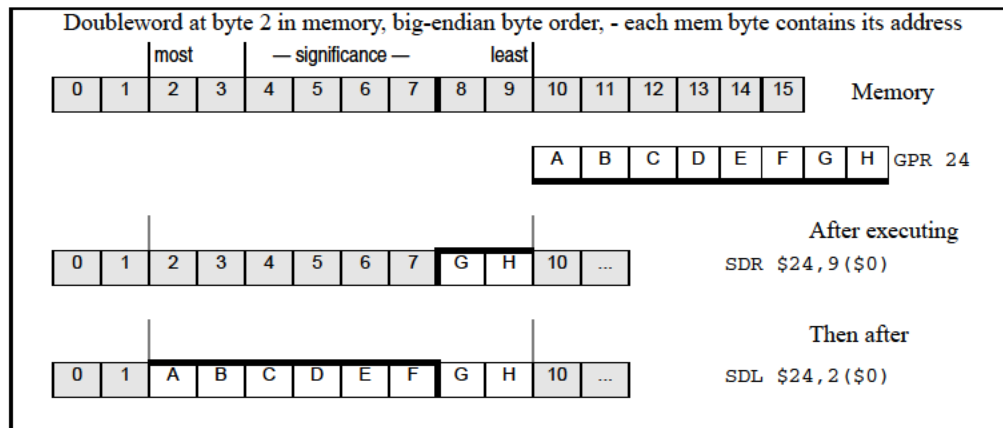
Description: $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{Some_Bytes_From } \text{GPR}[\text{rt}]$

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of 8 consecutive bytes forming a doubleword (*DW*) in memory, starting at an arbitrary byte boundary.

A part of *DW*, the least-significant 1 to 8 bytes, is in the aligned doubleword containing *EffAddr*. The same number of least-significant (right) bytes of GPR *rt* are stored into these bytes of *DW*.

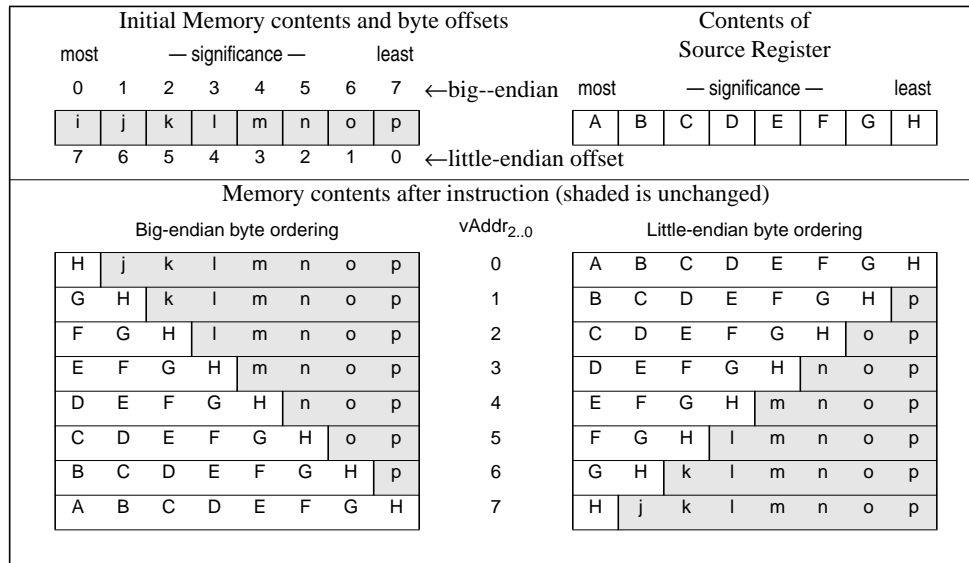
Figure 3-25 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 2 bytes, is located in the aligned doubleword containing the least-significant byte at 9. First, SDR stores the 2 least-significant bytes of the source register into these bytes in memory. Next, the complementary SDL stores the remainder of *DW*.

Figure 6.26 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 ($vAddr_{2..0}$)—and the current byte ordering mode of the processor (big- or little-endian). Figure 3-26 shows the bytes stored for every combination of offset and byte-ordering.

Figure 6.27 Bytes Stored by an SDR Instruction

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

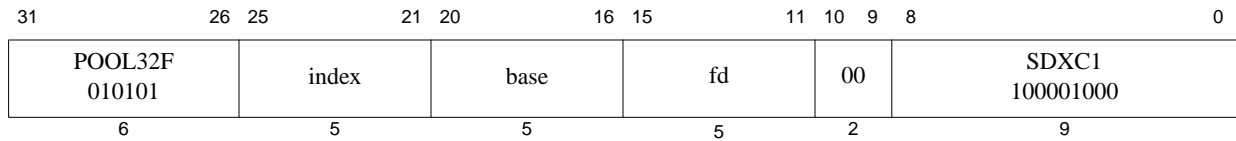
```

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



Format: SDXC1 *fd*, *index*(*base*)

microMIPS
microMIPS

Purpose: Store Doubleword Indexed from Floating Point

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

Description: $\text{memory}[\text{GPR}[\text{base}] + \text{GPR}[\text{index}]] \leftarrow \text{FPR}[\text{fd}]$

The 64-bit doubleword in FPR *fd* 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 $\text{EffectiveAddress}_{2..0} \neq 0$ (not doubleword-aligned).

Compatibility and Availability:

SDXC1: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode ($FIR_{F64}=0$ or 1, $Status_{FR}=0$ or 1).

Operation:

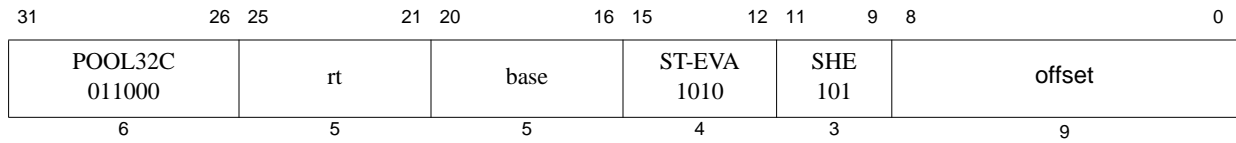
```
vAddr ← GPR[base] + GPR[index]
if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(fd, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Expected Instruction	Function	Equivalent Instruction
<code>ZEW rx, rx¹</code>	Zero-Extend Word	<code>DINSP32 rx, r0, 32, 32</code>

1. The equivalent instruction uses `rx` for both source and destination, so the expected instruction is limited to one register



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

microMIPS

Purpose: Store Halfword EVA

To store a halfword to user mode virtual address space when executing in kernel mode.

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

The least-significant 16-bit halfword of register *rt* is stored in memory at the location specified by the aligned effective address. The 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 ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian2 || 0))
bytesel ← vAddr2..0 xor (BigEndianCPU2 || 0)
datadoubleword ← GPR[rt]63-8*bytesel..0 || 08*bytesel
StoreMemory(CCA, HALFWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill

TLB Invalid

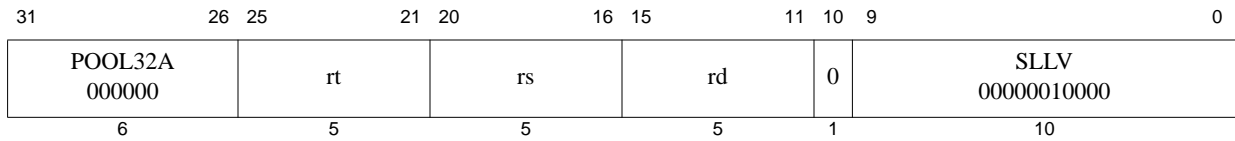
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



Format: SLLV rd, rt, rs

microMIPS

Purpose: Shift Word Left Logical Variable

To left-shift a word by a variable number of bits

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

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result word is sign-extended and placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

None

Operation:

```

s ← GPR[rs]4..0
temp ← GPR[rt](31-s)..0 || 0s
GPR[rd] ← sign_extend(temp)

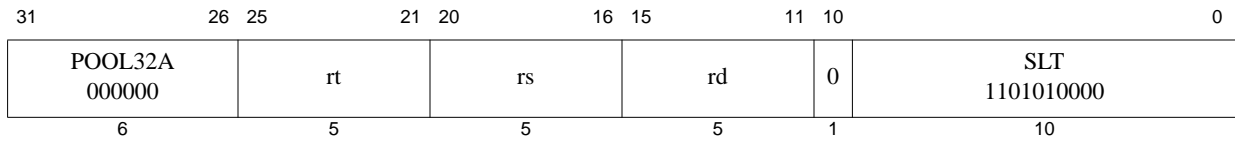
```

Exceptions:

None

Programming Notes:

Unlike nearly all other word operations, the input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign-extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign-extends it.



Format: SLT rd, rs, rt

microMIPS

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:

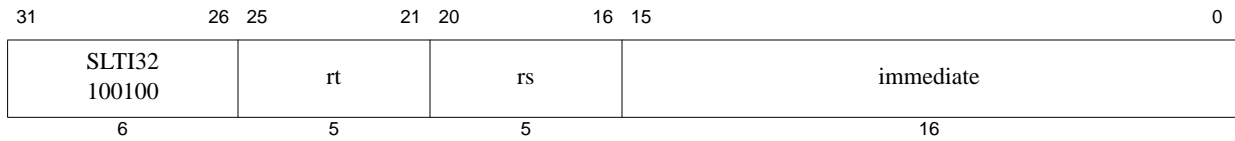
```

if GPR[rs] < GPR[rt] then
    GPR[rd] ← 0GPREN-1 || 1
else
    GPR[rd] ← 0GPREN
endif

```

Exceptions:

None



Format: SLTI *rt*, *rs*, *immediate*

microMIPS

Purpose: Set on Less Than Immediate

To record the result of a less-than comparison with a constant

Description: $GPR[rt] \leftarrow (GPR[rs] < immediate)$

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

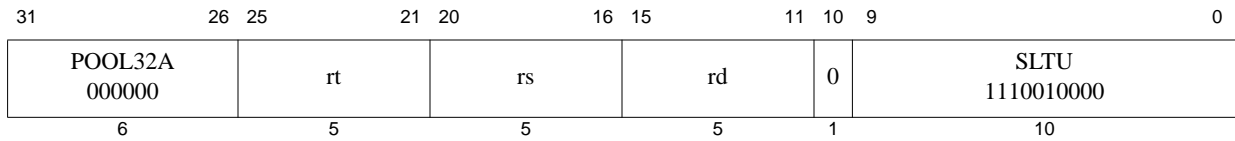
```

if GPR[rs] < sign_extend(immediate) then
    GPR[rt] ← 0GPREN-1 || 1
else
    GPR[rt] ← 0GPREN
endif

```

Exceptions:

None



Format: SLTU rd, rs, rt

microMIPS

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:

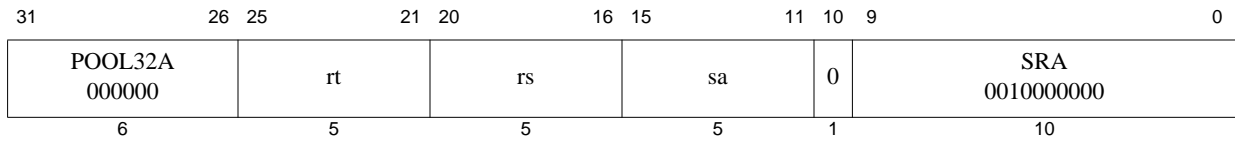
```

if (0 || GPR[rs]) < (0 || GPR[rt]) then
  GPR[rd] ← 0GPREN-1 || 1
else
  GPR[rd] ← 0GPREN
endif

```

Exceptions:

None



Format: SRA *rt*, *rs*, *sa*

microMIPS

Purpose: Shift Word Right Arithmetic

To execute an arithmetic right-shift of a word by a fixed number of bits

Description: $GPR[rt] \leftarrow GPR[rs] \gg sa$ (arithmetic)

The contents of the low-order 32-bit word of GPR *rs* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is sign-extended and placed in GPR *rt*. The bit-shift amount is specified by *sa*.

Restrictions:

On 64-bit processors, 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:

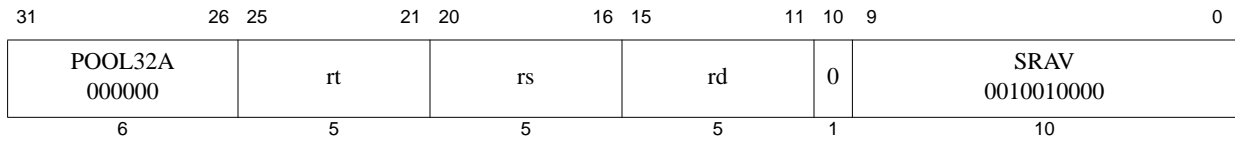
```

if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif
s ← sa
temp ← (GPR[rs]31)s || GPR[rs]31..s
GPR[rt] ← sign_extend(temp)

```

Exceptions:

None



Format: SRAV rd, rt, rs

microMIPS

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:

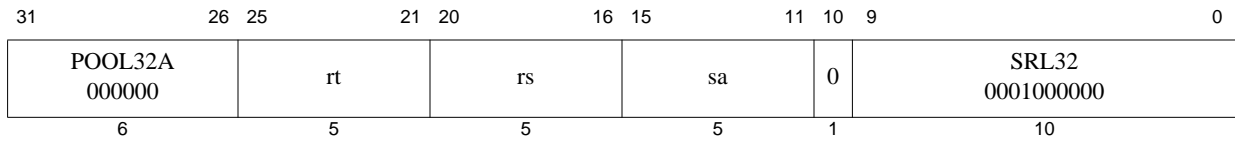
```

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



Format: SRL *rt*, *rs*, *sa*

microMIPS

Purpose: Shift Word Right Logical

To execute a logical right-shift of a word by a fixed number of bits

Description: $GPR[rt] \leftarrow GPR[rs] \gg sa$ (logical)

The contents of the low-order 32-bit word of GPR *rs* are shifted right, inserting zeros into the emptied bits; the word result is sign-extended and placed in GPR *rt*. The bit-shift amount is specified by *sa*.

Restrictions:

On 64-bit processors, 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:

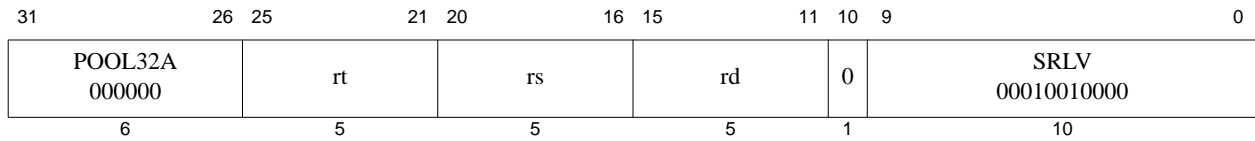
```

if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif
s ← sa
temp ← 0s || GPR[rs]31..s
GPR[rt] ← sign_extend(temp)

```

Exceptions:

None



Format: SRLV rd, rt, rs

microMIPS

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:

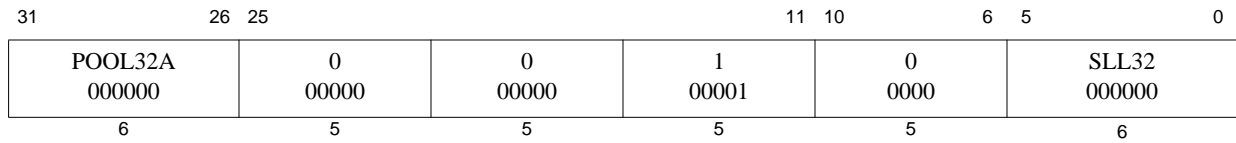
```

if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
s ← GPR[rs]4..0
temp ← 0s || GPR[rt]31..s
GPR[rd] ← sign_extend(temp)

```

Exceptions:

None



Format: SSNOP

microMIPS

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:

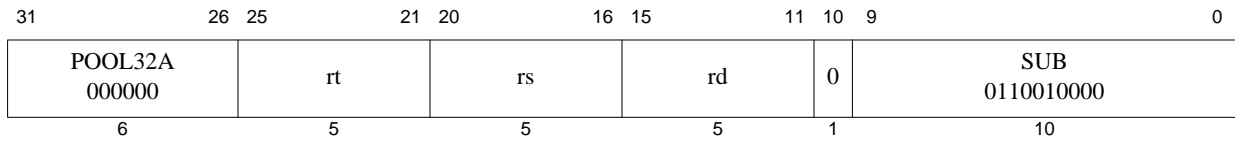
SSNOP is intended for use primarily to allow the programmer control over CPO hazards by converting instructions into cycles in a superscalar processor. For example, to insert at least two cycles between an MTC0 and an ERET, one would use the following sequence:

```

mtc0   x,y
ssnop
ssnop
eret

```

Based on the normal issues rules of the processor, the MTC0 issues in cycle T. Because the SSNOP instructions must issue alone, they may issue no earlier than cycle T+1 and cycle T+2, respectively. Finally, the ERET issues no earlier than cycle T+3. Note that although the instruction after an SSNOP may issue no earlier than the cycle after the SSNOP is issued, that instruction may issue later. This is because other implementation-dependent issue rules may apply that prevent an issue in the next cycle. Processors should not introduce any unnecessary delay in issuing SSNOP instructions.



Format: SUB rd, rs, rt

microMIPS

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

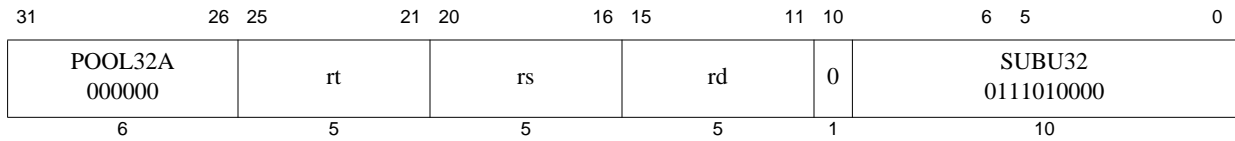
```

Exceptions:

Integer Overflow

Programming Notes:

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



Format: SUBU *rd*, *rs*, *rt*

microMIPS

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)

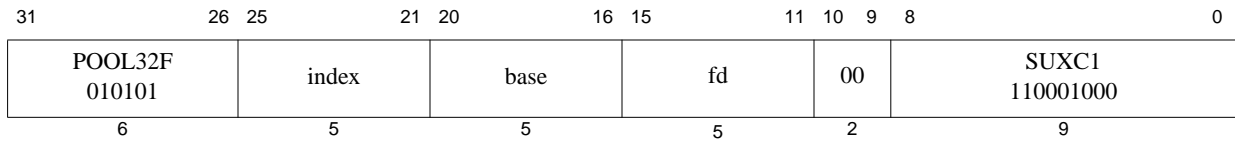
```

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.



Format: SUXC1 fd, index(base)

microMIPS

Purpose: Store Doubleword Indexed Unaligned from Floating Point

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

Description: $\text{memory}[(\text{GPR}[\text{base}] + \text{GPR}[\text{index}])_{\text{PSIZE}-1..3}] \leftarrow \text{FPR}[\text{fd}]$

The contents of the 64-bit doubleword in FPR *fd* is stored at the memory location specified by the effective address. The contents of GPR *index* and GPR *base* are added to form the effective address. The effective address is doubleword-aligned; EffectiveAddress_{2..0} are ignored.

Restrictions:

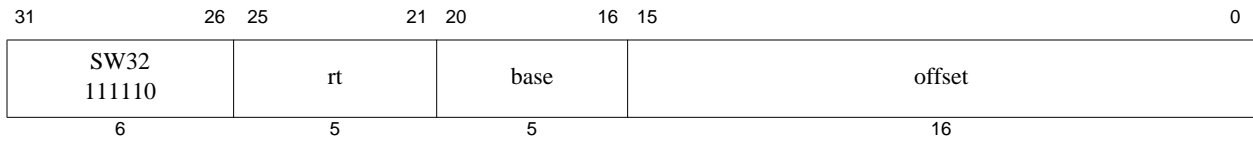
The result of this instruction is **UNPREDICTABLE** if the processor is executing in the *FR=0* 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the *FR=1* mode, but not with *FR=0*, and not on a 32-bit FPU.

Operation:

```
vAddr ← (GPR[base]+GPR[index])63..3 || 03
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(fd, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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



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

microMIPS

Purpose: Store Word

To store a word to memory

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

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

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

Operation:

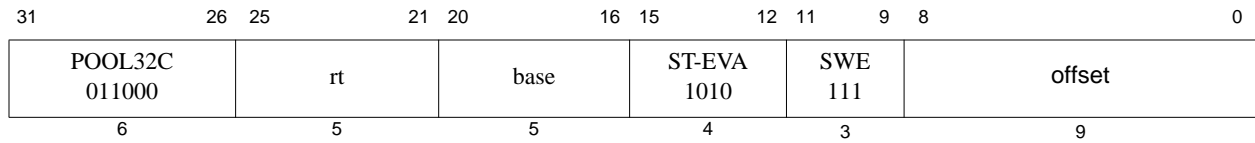
```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 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
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch



Format: SWE rt, offset(base)

microMIPS

Purpose: Store Word EVA

To store a word to user mode virtual address space when executing in kernel mode.

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

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

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

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

Restrictions:

Only usable in kernel mode when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

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

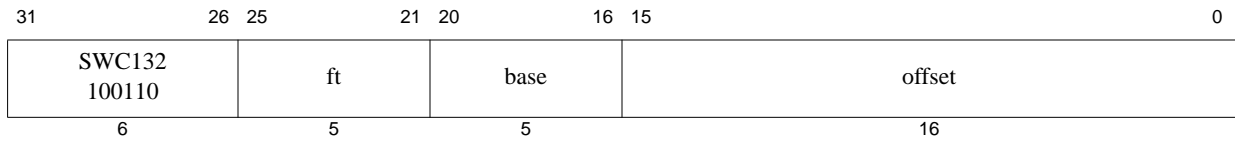
Bus Error

Address Error

Watch

Reserved Instruction

Coprocessor Unusable



SWC1 ft, offset(base)

microMIPS

Purpose: Store Word from Floating Point

To store a word from an FPR to memory

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

The low 32-bit word from FPR *ft* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

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

Operation:

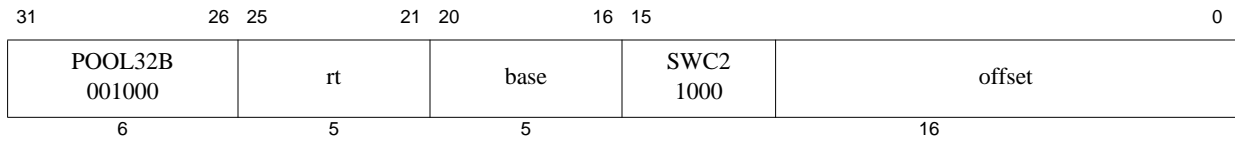
```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
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



Format: SWC2 *rt*, *offset*(*base*)

microMIPS

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 $\text{EffectiveAddress}_{1..0} \neq 0$ (not word-aligned).

Operation:

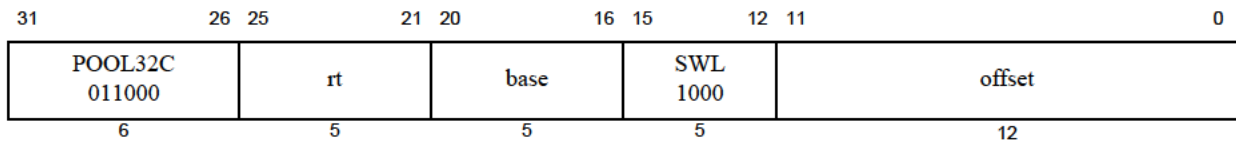
```

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



Format: SWL rt, offset(base)

microMIPS

Purpose: Store Word Left

To store the most-significant part of a word to an unaligned memory address

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

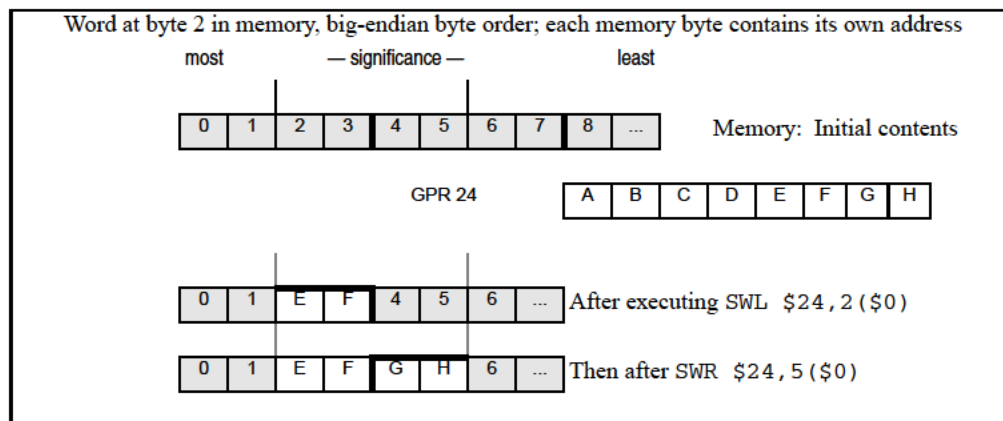
The 12-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

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.

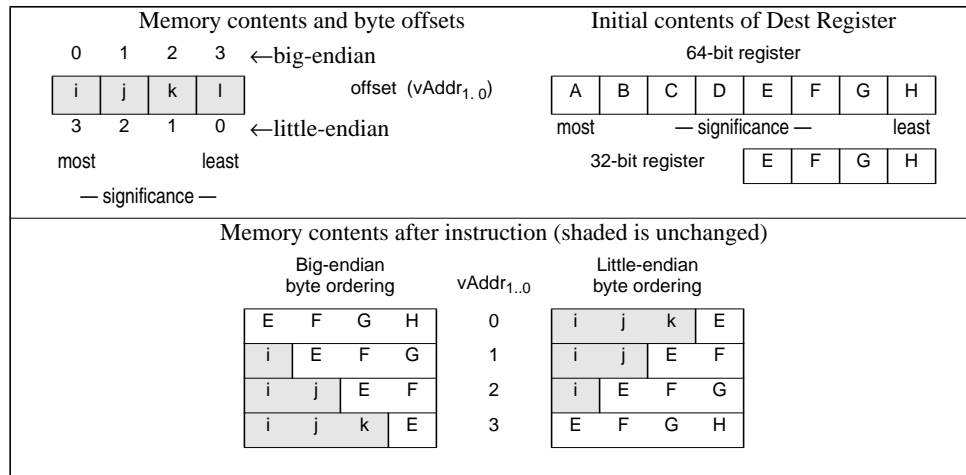
The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWL stores 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 6.28 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 6.29 Bytes Stored by an SWL Instruction

**Restrictions:**

None

Operation:

```

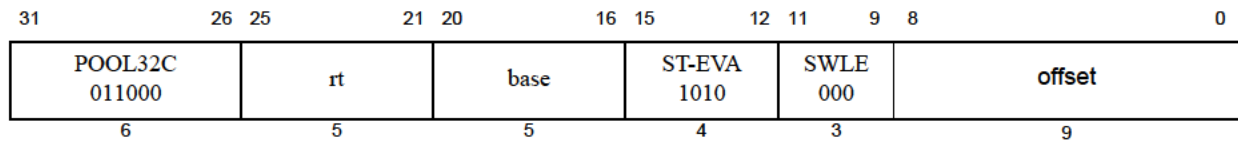
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
If BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..2 || 02
endif
byte ← vAddr1..0 xor BigEndianCPU2
if (vAddr2 xor BigEndianCPU) = 0 then
    datadoubleword ← 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



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

microMIPS

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

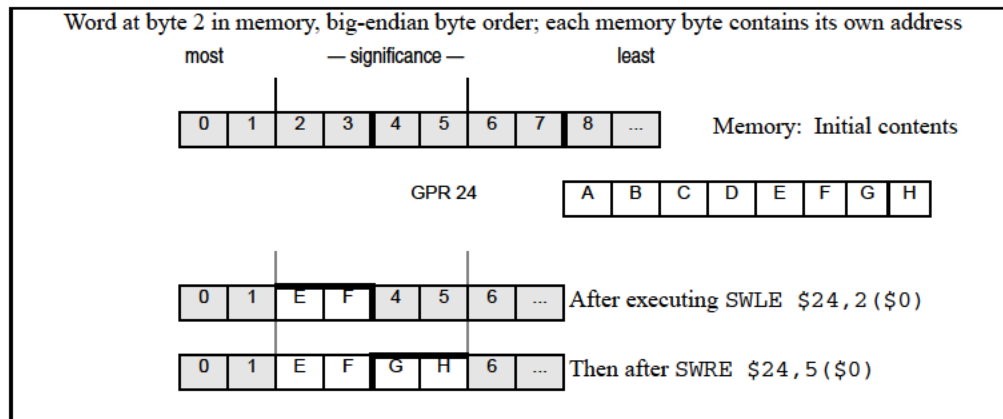
The 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

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 6.30 Unaligned Word Store Using SWLE and SWRE

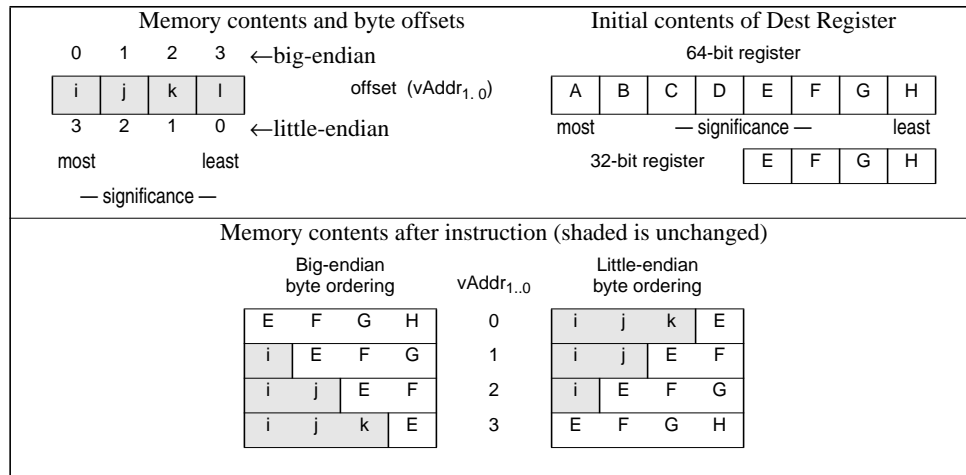


The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address ($vAddr_{1,0}$)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored 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 6.31 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:

```

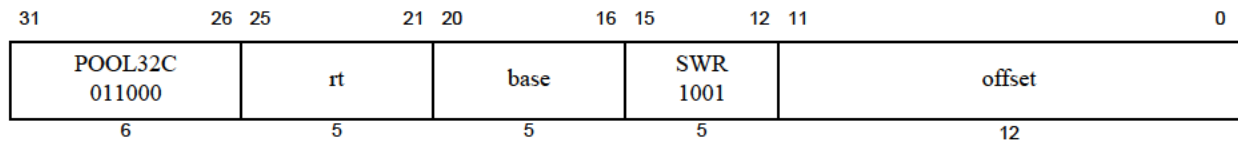
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
If BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..2 || 02
endif
byte ← vAddr1..0 xor BigEndianCPU2
if (vAddr2 xor BigEndianCPU) = 0 then
    datadoubleword ← 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, Reserved Instruction, Coprocessor Unusable



Format: SWR rt, offset(base)

microMIPS

Purpose: Store Word Right

To store the least-significant part of a word to an unaligned memory address

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

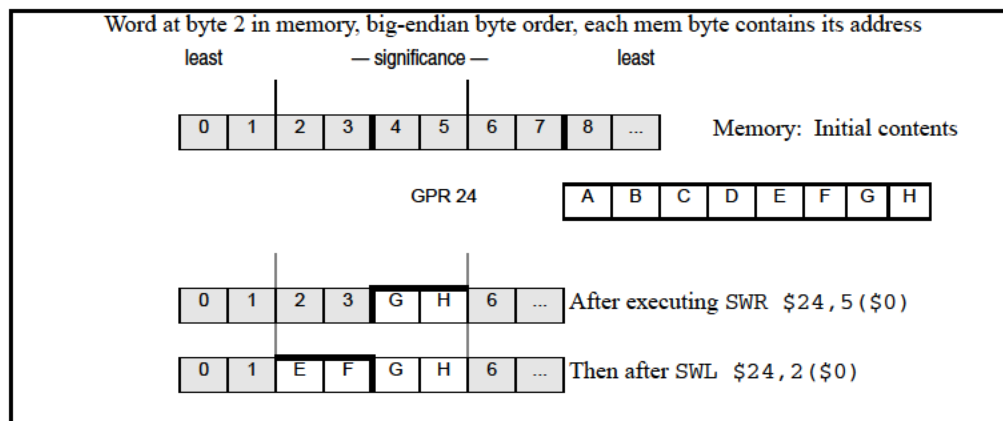
The 12-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

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.

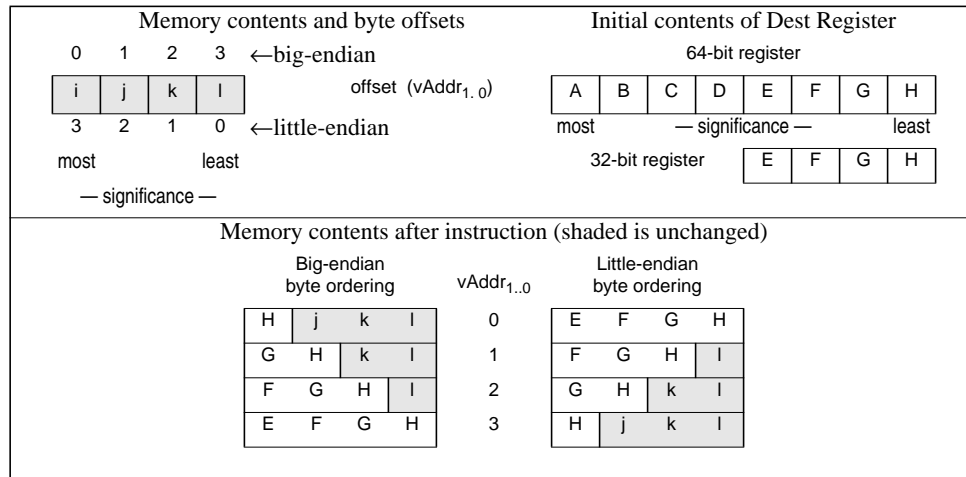
The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR 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 6.32 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 6.33 Bytes Stored by SWR Instruction

**Restrictions:**

None

Operation:

```

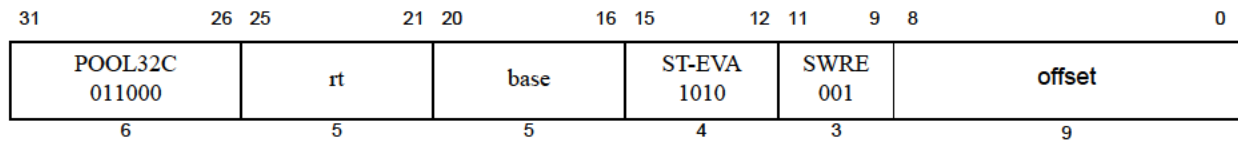
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
If BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..2 || 02
endif
byte ← vAddr1..0 xor BigEndianCPU2
if (vAddr2 xor BigEndianCPU) = 0 then
    datadoubleword ← 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



Format: SWRE rt, offset(base)

microMIPS

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

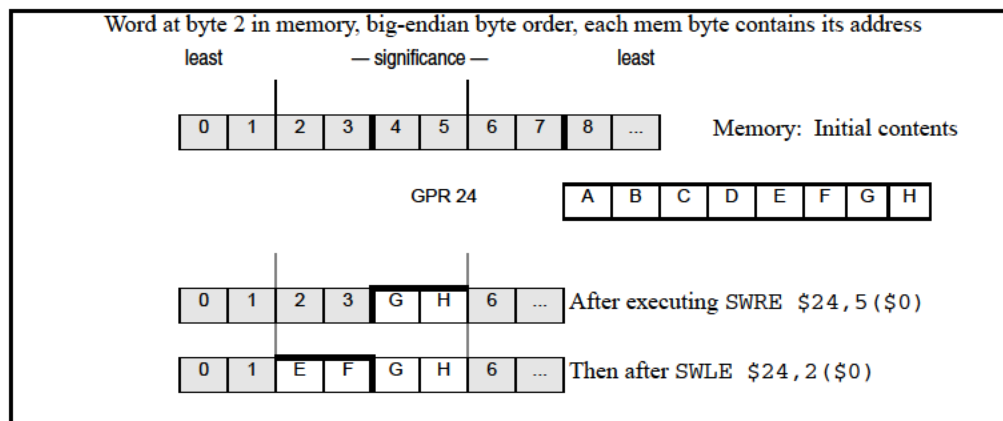
The 9-bit signed *offset* is added to the contents of GPR *base* to form an effective address (*EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word (*W*) in memory starting at an arbitrary byte boundary.

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 6.34 Unaligned Word Store Using SWRE and SWLE

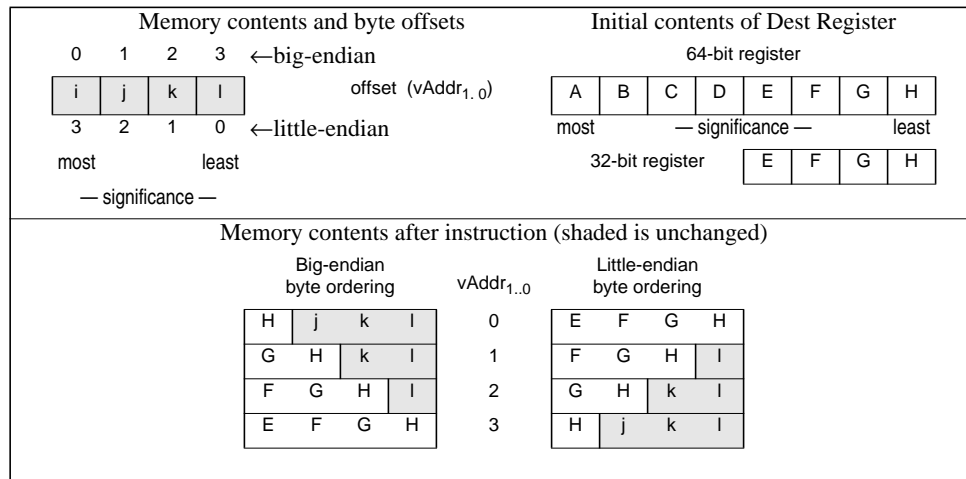


The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address ($vAddr_{1,0}$)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte-ordering.

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

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

Figure 6.35 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:

```

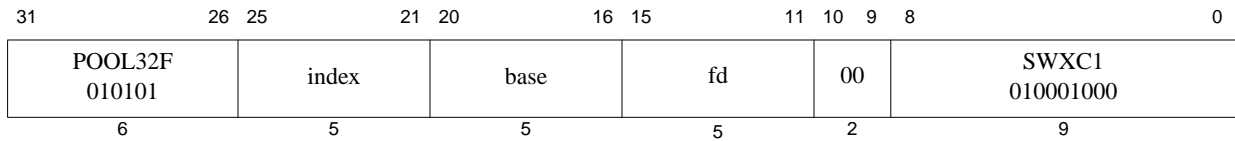
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
If BigEndianMem = 0 then
    pAddr ← pAddrPSIZE-1..2 || 02
endif
byte ← vAddr1..0 xor BigEndianCPU2
if (vAddr2 xor BigEndianCPU) = 0 then
    datadoubleword ← 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



Format: SWXC1 fd, index(base)

microMIPS
microMIPS

Purpose: Store Word Indexed from Floating Point

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

Description: $\text{memory}[\text{GPR}[\text{base}] + \text{GPR}[\text{index}]] \leftarrow \text{FPR}[\text{fd}]$

The low 32-bit word from FPR *fd* 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 $\text{EffectiveAddress}_{1..0} \neq 0$ (not word-aligned).

Compatibility and Availability:

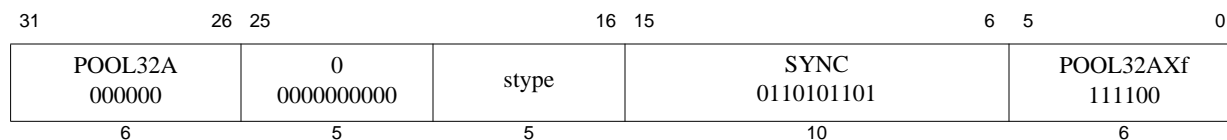
SWXC1: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode ($FIR_{F64}=0$ or 1, $FR=0$ or 1,)

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(fd, 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



Format: SYNC (styp = 0 implied)
 SYNC styp

microMIPS
 microMIPS

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 styp value of zero will always be defined such that it performs the most complete set of synchronization operations that are defined. This means styp 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 styp may be defined by the architecture or specific implementations to perform synchronization behaviors that are less complete than that of styp 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 styp must act the same as styp zero completion barrier. This allows software written for an implementation with a lighter-weight barrier to work on another implementation which only implements the styp zero completion barrier.
- A completion barrier is required, potentially in conjunction with SSNOP (in Release 1 of the Architecture) or EHB (in Release 2 of the Architecture), to guarantee that memory reference results are visible across operating mode changes. For example, a completion barrier is required on some implementations on entry to and exit from Debug Mode to guarantee that memory effects are handled correctly.

SYNC behavior when the stype field is zero:

- A completion barrier that affects preceding loads and stores and subsequent loads and stores.

Simple Description for Ordering Barrier:

- The barrier affects only *uncached* and *cached coherent* loads and stores.
- The specified memory instructions (loads or stores or both) that occur before the SYNC instruction must always be ordered before the specified memory instructions after the SYNC.
- Memory instructions which are ordered before other memory instructions are processed by the load/store datapath first before the other memory instructions.

Detailed Description for Ordering Barrier:

- Every synchronizable specified memory instruction (loads or stores or both) that occurs in the instruction stream before the SYNC instruction must reach a stage in the load/store datapath after which no instruction re-ordering is possible before any synchronizable specified memory instruction which occurs after the SYNC instruction in the instruction stream reaches the same stage in the load/store datapath.
- If any memory instruction before the SYNC instruction in program order, generates a memory request to the external memory and any memory instruction after the SYNC instruction in program order also generates a memory request to external memory, the memory request belonging to the older instruction must be globally performed before the time the memory request belonging to the younger instruction is globally performed.
- The barrier does not guarantee the order in which instruction fetches are performed.

As compared to the completion barrier, the ordering barrier is a lighter-weight operation as it does not require the specified instructions before the SYNC to be already completed. Instead it only requires that those specified instructions which are subsequent to the SYNC in the instruction stream are never re-ordered for processing ahead of the specified instructions which are before the SYNC in the instruction stream. This potentially reduces how many cycles the barrier instruction must stall before it completes.

The Acquire and Release barrier types are used to minimize the memory orderings that must be maintained and still have software synchronization work.

Implementations that do not use any of the non-zero values of stype to define different barriers, such as ordering barriers, must make those stype values act the same as stype zero.

For the purposes of this description, the CACHE, PREF and PREFX instructions are treated as loads and stores. That is, these instructions and the memory transactions sourced by these instructions obey the ordering and completion rules of the SYNC instruction.

Table 6.28 lists the available completion barrier and ordering barriers behaviors that can be specified using the stype field..

Table 6.28 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 previ-

ously made to the same physical location using a different memory access type. The presence of a SYNC between the references does not alter this behavior.

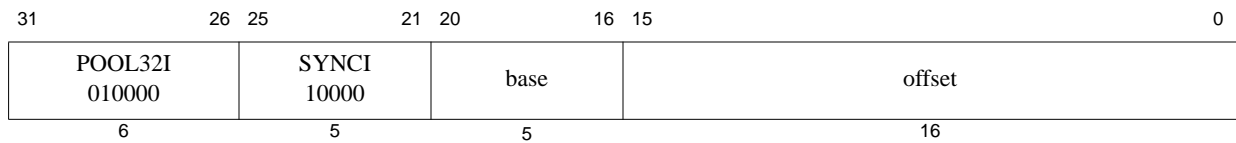
SYNC affects the order in which the effects of load and store instructions appear to all processors; it does not generally affect the physical memory-system ordering or synchronization issues that arise in system programming. The effect of SYNC on implementation-specific aspects of the cached memory system, such as writeback buffers, is not defined.

```
# Processor A (writer)
# Conditions at entry:
# The value 0 has been stored in FLAG and that value is observable by B
SW    R1, DATA      # change shared DATA value
LI    R2, 1
SYNC                      # Perform DATA store before performing FLAG store
SW    R2, FLAG       # say that the shared DATA value is valid

# Processor B (reader)
LI    R2, 1
1: LW  R1, FLAG      # Get FLAG
BNE   R2, R1, 1B    # if it says that DATA is not valid, poll again
NOP
SYNC                      # FLAG value checked before doing DATA read
LW    R1, DATA     # Read (valid) shared DATA value
```

The code fragments above shows how SYNC can be used to coordinate the use of shared data between separate writer and reader instruction streams in a multiprocessor environment. The FLAG location is used by the instruction streams to determine whether the shared data item DATA is valid. The SYNC executed by processor A forces the store of DATA to be performed globally before the store to FLAG is performed. The SYNC executed by processor B ensures that DATA is not read until after the FLAG value indicates that the shared data is valid.

Software written to use a SYNC instruction with a non-zero stype value, expecting one type of barrier behavior, should only be run on hardware that actually implements the expected barrier behavior for that non-zero stype value or on hardware which implements a superset of the behavior expected by the software for that stype value. If the hardware does not perform the barrier behavior expected by the software, the system may fail.



Format: SYNCI offset (base)

microMIPS

Purpose: Synchronize Caches to Make Instruction Writes Effective

To synchronize all caches to make instruction writes effective.

Description:

This instruction is used after a new instruction stream is written to make the new instructions effective relative to an instruction fetch, when used in conjunction with the SYNC and JALR.HB, JR.HB, or ERET instructions, as described below. Unlike the CACHE instruction, the SYNCI instruction is available in all operating modes in an implementation of Release 2 of the architecture.

The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used to address the cache line in all caches which may need to be synchronized with the write of the new instructions. The operation occurs only on the cache line which may contain the effective address. One SYNCI instruction is required for every cache line that was written. See the Programming Notes below.

A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur as a byproduct of this instruction. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS. This instruction never causes Execute-Inhibit nor Read-Inhibit exceptions.

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

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

It is implementation dependent whether a data watch is triggered by a SYNCI instruction whose address matches the Watch register address match conditions. In multiprocessor implementations where instruction caches are not coherently maintained by hardware, the SYNCI instruction may optionally affect all coherent icaches within the system. If the effective address uses a coherent Cacheability and Coherency Attribute (CCA), then the operation may be *globalized*, meaning it is broadcast to all of the coherent instruction caches within the system. If the effective address does not use one of the coherent CCAs, there is no broadcast of the SYNCI operation. If multiple levels of caches are to be affected by one SYNCI instruction, all of the affected cache levels must be processed in the same manner - either all affected cache levels use the globalized behavior or all affected cache levels use the non-globalized behavior.

In multiprocessor implementations where instruction caches are coherently maintained by hardware, the SYNCI instruction should behave as a NOP instruction.

Restrictions:

The operation of the processor is **UNPREDICTABLE** if the effective address references any instruction cache line that contains instructions to be executed between the SYNCI and the subsequent JALR.HB, JR.HB, or ERET instruction required to clear the instruction hazard.

The SYNCI instruction has no effect on cache lines that were previously locked with the CACHE instruction. If correct software operation depends on the state of a locked line, the CACHE instruction must be used to synchronize the caches.

The SYNCI instruction acts on the current processor at a minimum. It is implementation specific whether it affects

the caches on other processors in a multiprocessor system, except as required to perform the operation on the current processor (as might be the case if multiple processors share an L2 or L3 cache).

Full visibility of the new instruction stream requires execution of a subsequent SYNC instruction, followed by a JALR.HB, JR.HB, DERET, or ERET instruction. The operation of the processor is **UNPREDICTABLE** if this sequence is not followed.

Operation:

```
vaddr ← GPR[base] + sign_extend(offset)
SynchronizeCacheLines(vaddr)      /* Operate on all caches */
```

Exceptions:

Reserved Instruction Exception (Release 1 implementations only)

TLB Refill Exception

TLB Invalid Exception

Address Error Exception

Cache Error Exception

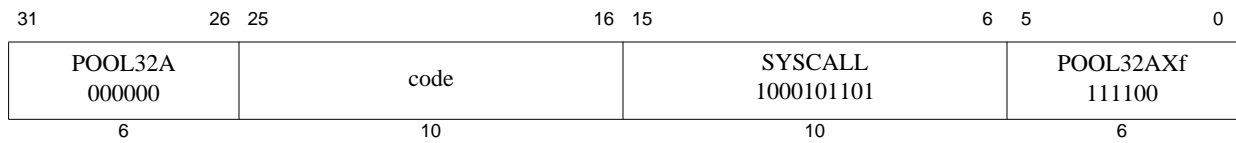
Bus Error Exception

Programming Notes:

When the instruction stream is written, the SYNCI instruction should be used in conjunction with other instructions to make the newly-written instructions effective. The following example shows a routine which can be called after the new instruction stream is written to make those changes effective. Note that the SYNCI instruction could be replaced with the corresponding sequence of CACHE instructions (when access to Coprocessor 0 is available), and that the JR.HB instruction could be replaced with JALR.HB, ERET, or DERET instructions, as appropriate. A SYNC instruction is required between the final SYNCI instruction in the loop and the instruction that clears instruction hazards.

```
/*
 * This routine makes changes to the instruction stream effective to the
 * hardware. It should be called after the instruction stream is written.
 * On return, the new instructions are effective.
 *
 * Inputs:
 *   a0 = Start address of new instruction stream
 *   a1 = Size, in bytes, of new instruction stream
 */

    beq    a1, zero, 20f      /* If size==0, */
    nop                    /* branch around */
    addu   a1, a0, a1        /* Calculate end address + 1 */
                                /* (daddu for 64-bit addressing) */
    rdhwr  v0, HW_SYNCI_Step /* Get step size for SYNCI from new */
                                /* Release 2 instruction */
    beq    v0, zero, 20f    /* If no caches require synchronization, */
    nop                    /* branch around */
10: synci 0(a0)             /* Synchronize all caches around address */
    addu   a0, a0, v0        /* Add step size in delay slot */
                                /* (daddu for 64-bit addressing) */
    sltu   v1, a0, a1        /* Compare current with end address */
    bne    v1, zero, 10b    /* Branch if more to do */
    nop                    /* branch around */
    sync                                /* Clear memory hazards */
20: jr.hb ra                /* Return, clearing instruction hazards */
    nop
```

Format: SYSCALL

microMIPS

Purpose: System Call

To cause a System Call exception

Description:

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Restrictions:

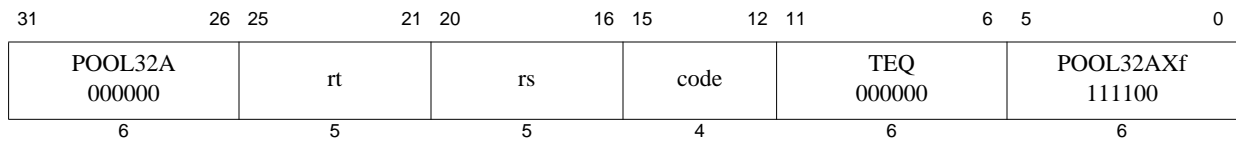
None

Operation:

`SignalException(SystemCall)`

Exceptions:

System Call



Format: TEQ *rs*, *rt*

microMIPS

Purpose: Trap if Equal

To compare GPRs and do a conditional trap

Description: if GPR[*rs*] = GPR[*rt*] then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

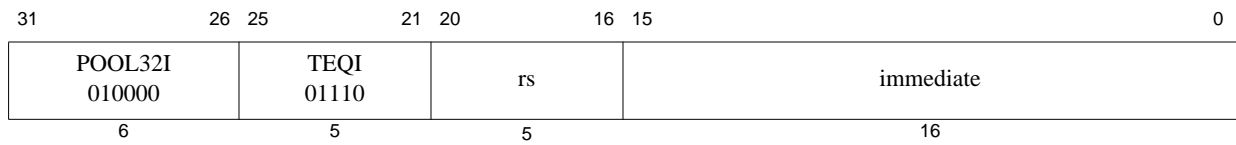
None

Operation:

```
if GPR[rs] = GPR[rt] then
    SignalException(Trap)
endif
```

Exceptions:

Trap



Format: TEQI rs, immediate

microMIPS

Purpose: Trap if Equal Immediate

To compare a GPR to a constant and do a conditional trap

Description: if GPR[rs] = immediate then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is equal to *immediate*, then take a Trap exception.

Restrictions:

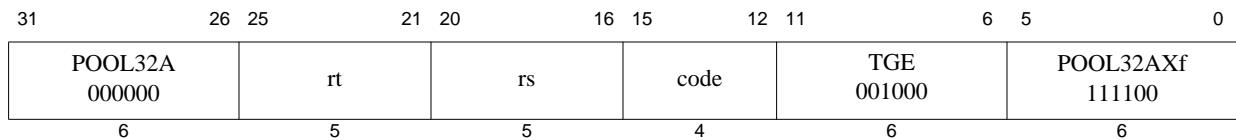
None

Operation:

```
if GPR[rs] = sign_extend(immediate) then
    SignalException(Trap)
endif
```

Exceptions:

Trap



Format: TGE *rs*, *rt*

microMIPS

Purpose: Trap if Greater or Equal

To compare GPRs and do a conditional trap

Description: if $GPR[rs] \geq GPR[rt]$ then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is greater than or equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

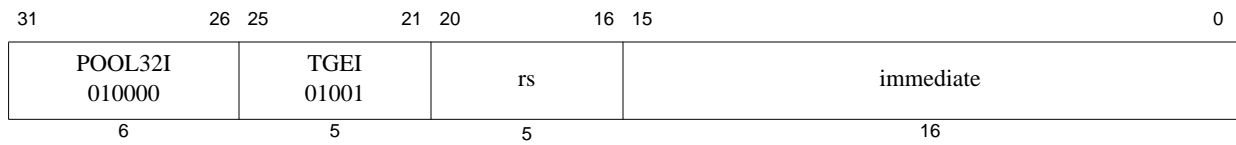
None

Operation:

```
if GPR[rs] ≥ GPR[rt] then
    SignalException(Trap)
endif
```

Exceptions:

Trap



Format: TGEI rs, immediate

microMIPS

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 Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is greater than or equal to *immediate*, then take a Trap exception.

Restrictions:

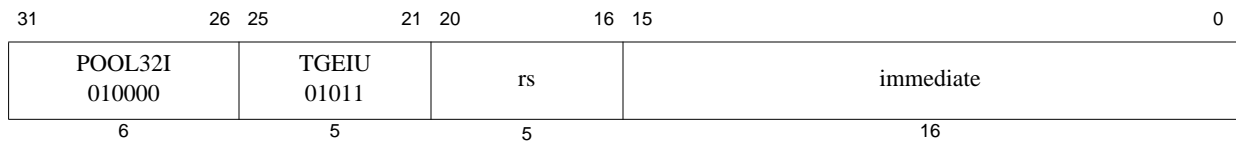
None

Operation:

```
if GPR[rs] ≥ sign_extend(immediate) then
    SignalException(Trap)
endif
```

Exceptions:

Trap



Format: TGEIU *rs*, *immediate*

microMIPS

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 Trap

Compare the contents of GPR *rs* and the 16-bit sign-extended *immediate* as unsigned integers; if GPR *rs* is greater than or equal to *immediate*, then take a Trap exception.

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

Restrictions:

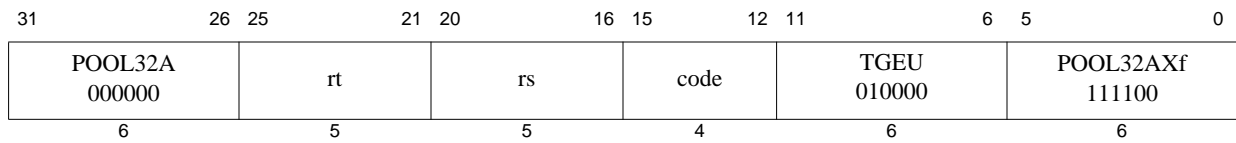
None

Operation:

```
if (0 || GPR[rs]) ≥ (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

Exceptions:

Trap



Format: TGEU *rs*, *rt*

microMIPS

Purpose: Trap if Greater or Equal Unsigned

To compare GPRs and do a conditional trap

Description: if $GPR[rs] \geq GPR[rt]$ then Trap

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; if GPR *rs* is greater than or equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

```
if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
    SignalException(Trap)
endif
```

Exceptions:

Trap

31	26	25	16	15	6	5	0
POOL32A 000000	0 0000000000		TLBP 0000001101		POOL32AXf 111100		
6	10		10		6		

Format: TLBP

microMIPS

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	16	15	6	5	0
POOL32A 000000	0 0000000000		TLBR 0001001101		POOL32AXf 111100		
6	10		10		6		6

Format: TLBR

microMIPS

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

```

```

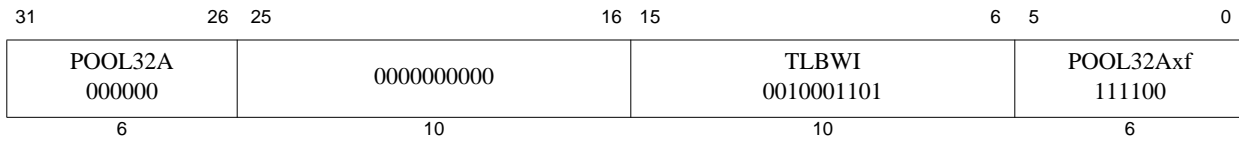
EntryLo0 ← 0
EntryHiEHINV ← 1
else
PageMaskMask ← TLB[i]Mask
EntryHi ← TLB[i]R || 0Fill ||
          (TLB[i]VPN2 and not TLB[i]Mask) || # Masking implem dependent
          05 || TLB[i]ASID
EntryLo1 ← 0Fill ||
          (TLB[i]PFN1 and not TLB[i]Mask) || # Masking mplem dependent
          TLB[i]C1 || 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:

Coprocesor Unusable

Machine Check



Format: TLBWI

microMIPS

Purpose: Write Indexed TLB Entry

To write or invalidate a TLB entry indexed by the *Index* register.

Description:

If $Config4_{IE} < 2$ or $EntryHi_{EHINV} = 0$:

The TLB entry pointed to by the Index register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers. It is implementation dependent whether multiple TLB matches are detected on a TLBWI. In such an instance, a Machine Check Exception is signaled. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the *PageMask* register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of *PageMask* register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The single G bit in the TLB entry is set from the logical AND of the *G* bits in the *EntryLo0* and *EntryLo1* registers.

If $Config4_{IE} > 1$ and $EntryHi_{EHINV} = 1$:

The TLB entry pointed to by the Index register has its VPN2 field marked as invalid. This causes the entry to be ignored on TLB matches for memory accesses. No Machine Check is generated.

Restrictions:

The operation is **UNDEFINED** if the contents of the Index register are greater than or equal to the number of TLB entries in the processor.

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

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
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:

Coproprocessor Unusable

Machine Check

31	26 25	16 15	6 5	0
POOL32A 000000	0000000000	TLBWR 0011001101	POOL32Axf 111100	
6	10	10	6	

Format: TLBWR

microMIPS

Purpose: Write Random TLB Entry

To write a TLB entry indexed by the *Random* register.

Description:

The TLB entry pointed to by the *Random* register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers. It is implementation dependent whether multiple TLB matches are detected on a TLBWR. In such an instance, a Machine Check Exception is signaled. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the *PageMask* register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of *PageMask* register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.

Restrictions:

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

Operation:

```

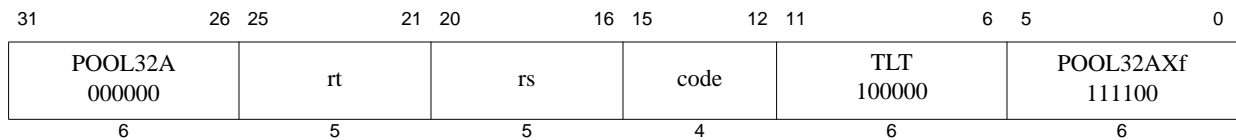
i ← Random
if (Config4TE = 2 or Config4TE = 3) then
    TLB[i]VPN2_invalid ← 0
endif
TLB[i]Mask ← PageMaskMask
TLB[i]R ← EntryHiR
TLB[i]VPN2 ← EntryHiVPN2 and not PageMaskMask # Implementation dependent
TLB[i]ASID ← EntryHiASID
TLB[i]G ← EntryLo1G and EntryLo0G
TLB[i]PFN1 ← EntryLo1PFN and not PageMaskMask # Implementation dependent
TLB[i]C1 ← EntryLo1C
TLB[i]D1 ← EntryLo1D
TLB[i]V1 ← EntryLo1V
TLB[i]PFN0 ← EntryLo0PFN and not PageMaskMask # Implementation dependent
TLB[i]C0 ← EntryLo0C
TLB[i]D0 ← EntryLo0D
TLB[i]V0 ← EntryLo0V

```

Exceptions:

Coprocessor Unusable

Machine Check



Format: TLT *rs*, *rt*

microMIPS

Purpose: Trap if Less Than

To compare GPRs and do a conditional trap

Description: if GPR[*rs*] < GPR[*rt*] then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is less than GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

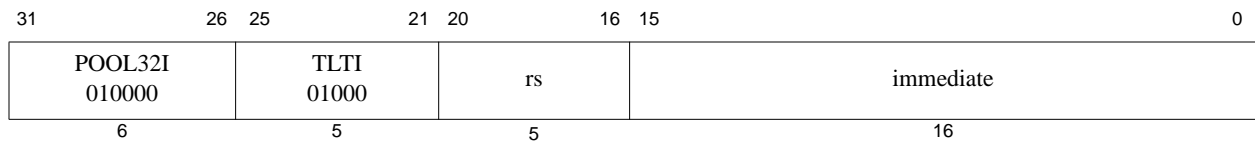
None

Operation:

```
if GPR[rs] < GPR[rt] then
    SignalException(Trap)
endif
```

Exceptions:

Trap



Format: TLTI rs, immediate

microMIPS

Purpose: Trap if Less Than Immediate

To compare a GPR to a constant and do a conditional trap

Description: if GPR[rs] < immediate then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is less than *immediate*, then take a Trap exception.

Restrictions:

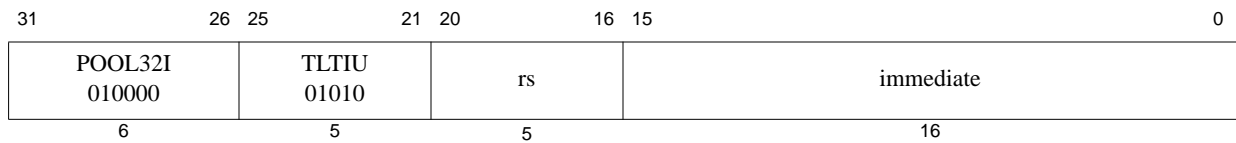
None

Operation:

```
if GPR[rs] < sign_extend(immediate) then
    SignalException(Trap)
endif
```

Exceptions:

Trap



Format: TLTIU *rs*, *immediate*

microMIPS

Purpose: Trap if Less Than Immediate Unsigned

To compare a GPR to a constant and do a conditional trap

Description: if $GPR[rs] < immediate$ then Trap

Compare the contents of GPR *rs* and the 16-bit sign-extended *immediate* as unsigned integers; if GPR *rs* is less than *immediate*, then take a Trap exception.

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

Restrictions:

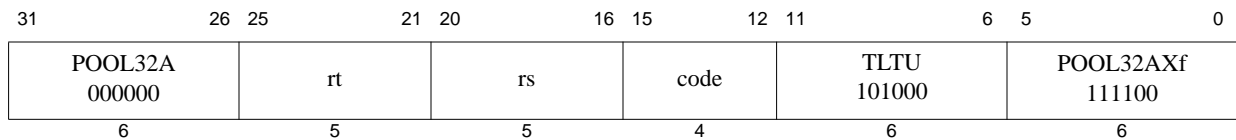
None

Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

Exceptions:

Trap



Format: TLTU *rs*, *rt*

microMIPS

Purpose: Trap if Less Than Unsigned

To compare GPRs and do a conditional trap

Description: if GPR[*rs*] < GPR[*rt*] then Trap

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; if GPR *rs* is less than GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

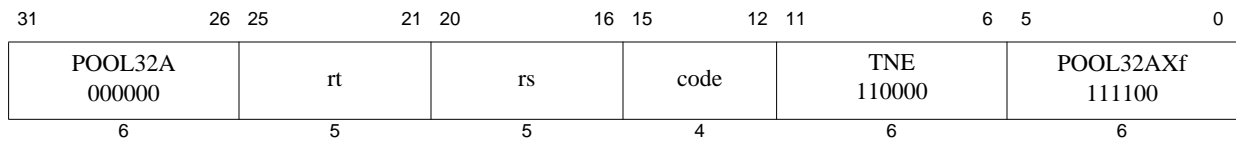
None

Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
    SignalException(Trap)
endif
```

Exceptions:

Trap



Format: TNE *rs*, *rt*

microMIPS

Purpose: Trap if Not Equal

To compare GPRs and do a conditional trap

Description: if GPR[*rs*] \neq GPR[*rt*] then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is not equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

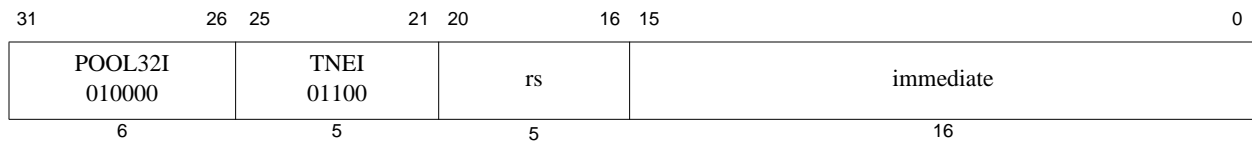
None

Operation:

```
if GPR[rs]  $\neq$  GPR[rt] then
    SignalException(Trap)
endif
```

Exceptions:

Trap



Format: TNEI rs, immediate

microMIPS

Purpose: Trap if Not Equal Immediate

To compare a GPR to a constant and do a conditional trap

Description: if GPR[rs] \neq immediate then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is not equal to *immediate*, then take a Trap exception.

Restrictions:

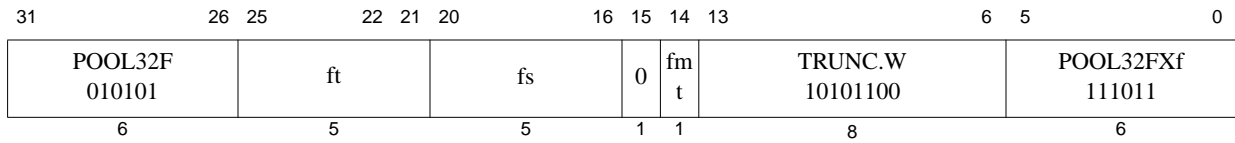
None

Operation:

```
if GPR[rs]  $\neq$  sign_extend(immediate) then
    SignalException(Trap)
endif
```

Exceptions:

Trap



Format: TRUNC.W.fmt
 TRUNC.W.S ft, fs
 TRUNC.W.D ft, fs

microMIPS
 microMIPS

Purpose: Floating Point Truncate to Word Fixed Point

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

Description: $FPR[ft] \leftarrow \text{convert_and_round}(FPR[fs])$

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 *ft*.

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 *ft* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *ft*.

Restrictions:

The fields *fs* and *ft* 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:

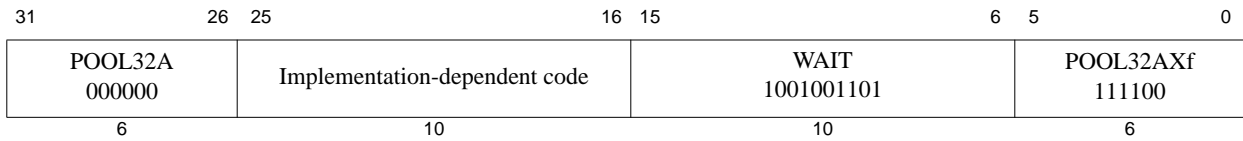
`StoreFPR(ft, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))`

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Invalid Operation, Unimplemented Operation



Format: WAIT

microMIPS

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.

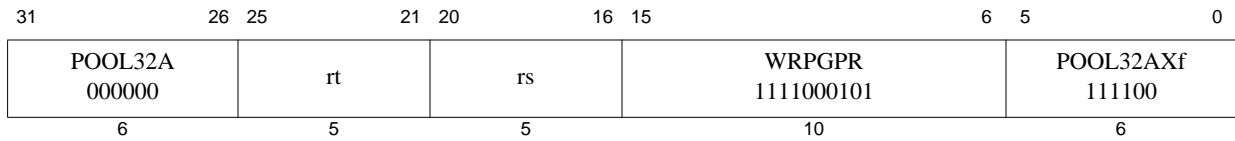
If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
I: Enter implementation dependent lower power mode
I+1: /* Potential interrupt taken here */
```

Exceptions:

Coprocessor Unusable Exception



Format: WRPGPR *rt*, *rs*

microMIPS

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: $SGPR[SRSCtl_{PSS}, rt] \leftarrow GPR[rs]$

The contents of the current GPR *rs* is moved to the shadow GPR register specified by $SRSCtl_{PSS}$ (signifying the previous shadow set number) and *rt* (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.

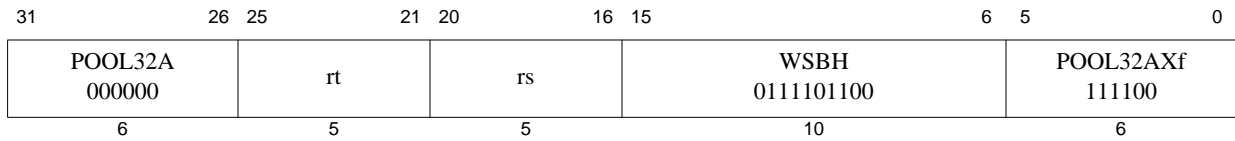
Operation:

$SGPR[SRSCtl_{PSS}, rt] \leftarrow GPR[rs]$

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: WSBH *rt*, *rs*

microMIPS

Purpose: Word Swap Bytes Within Halfwords

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

Description: $GPR[rt] \leftarrow \text{SwapBytesWithinHalfwords}(GPR[rs])$

Within each halfword of the lower word of GPR *rs* the bytes are swapped, the result is sign-extended, and stored in GPR *rt*.

Restrictions:

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

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

Operation:

```
if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
GPR[rt] ← sign_extend(GPR[rs]23..16 || GPR[rs]31..24 || GPR[rs]7..0 || GPR[rs]15..8)
```

Exceptions:

Reserved Instruction

Programming Notes:

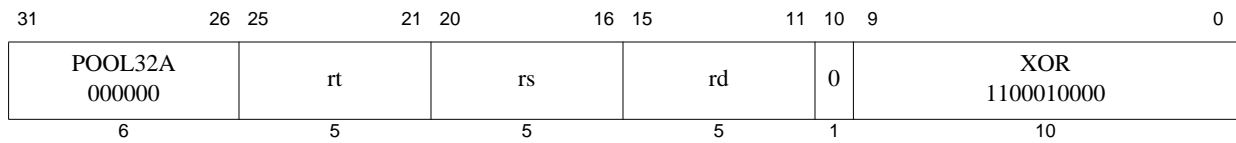
The WSBH instruction can be used to convert halfword and word data of one endianness to another endianness. The endianness of a word value can be converted using the following sequence:

```
lw    t0, 0(a1)           /* Read word value */
wsbh  t0, t0             /* Convert endiannes of the halfwords */
rotr  t0, t0, 16        /* Swap the halfwords within the words */
```

Combined with SEH and SRA, two contiguous halfwords can be loaded from memory, have their endianness converted, and be sign-extended into two word values in four instructions. For example:

```
lw    t0, 0(a1)           /* Read two contiguous halfwords */
wsbh  t0, t0             /* Convert endiannes of the halfwords */
seh   t1, t0             /* t1 = lower halfword sign-extended to word */
sra   t0, t0, 16        /* t0 = upper halfword sign-extended to word */
```

Zero-extended words can be created by changing the SEH and SRA instructions to ANDI and SRL instructions, respectively.



Format: XOR rd, rs, rt

microMIPS

Purpose: Exclusive OR

To do a bitwise logical Exclusive OR

Description: $GPR[rd] \leftarrow GPR[rs] \text{ XOR } GPR[rt]$

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical Exclusive OR operation and place the result into GPR *rd*.

Restrictions:

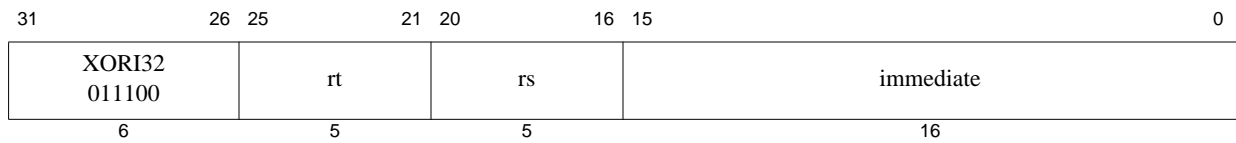
None

Operation:

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

Exceptions:

None



Format: XORI *rt*, *rs*, *immediate*

microMIPS

Purpose: Exclusive OR Immediate

To do a bitwise logical Exclusive OR with a constant

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

Combine the contents of GPR *rs* and the 16-bit zero-extended *immediate* in a bitwise logical Exclusive OR operation and place the result into GPR *rt*.

Restrictions:

None

Operation:

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

Exceptions:

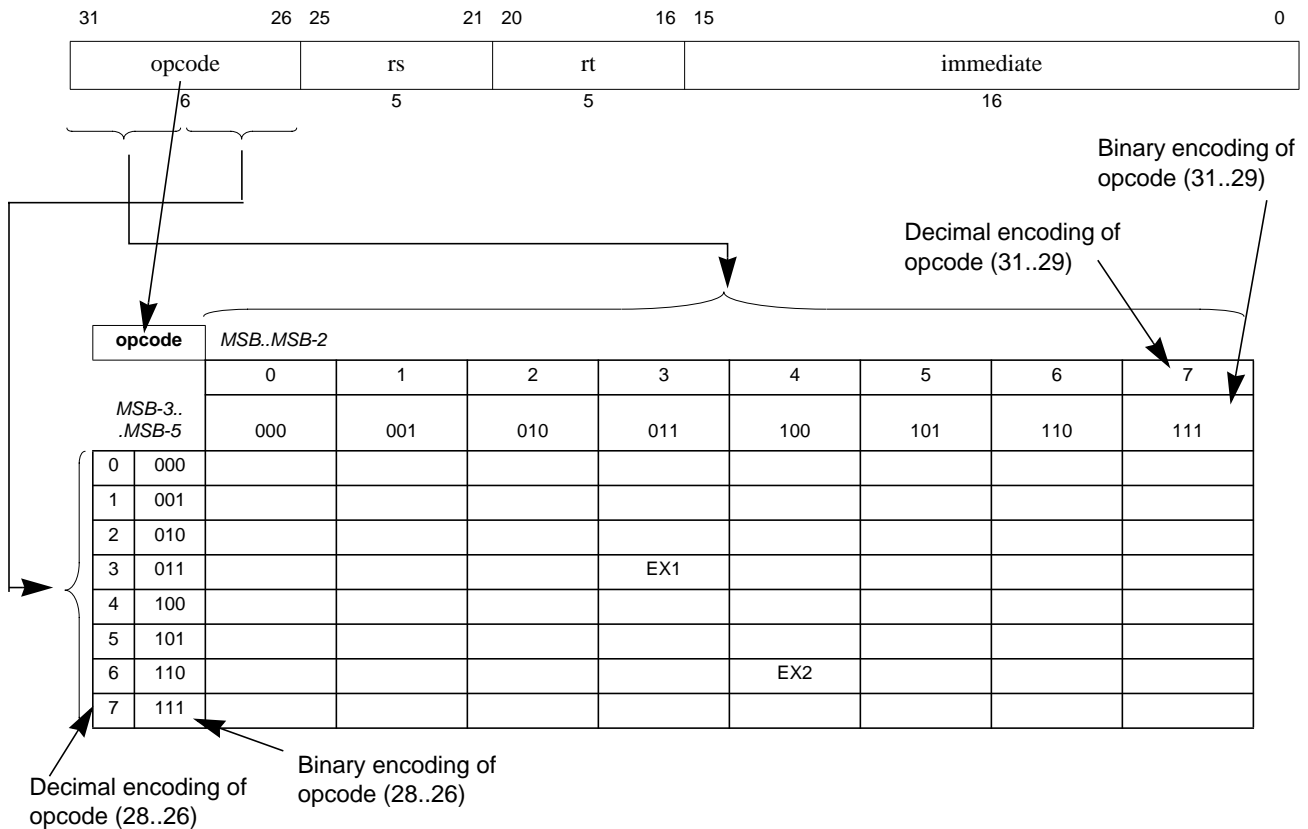
None

Opcode Map

This chapter defines the bit-level encoding of all microMIPS64 instructions, using a series of opcode tables. The basic format of the tables is shown in Figure 7.1. The topmost row contains the high-order opcode bits (in the example table shown here, bits 31..29), and the left-most column of the table lists the next most-significant bits of the opcode field (bits 28..26). Decimal and binary values are shown for both rows and columns.

An instruction's encoding is the value at the intersection of a row and column. For example, the opcode value for the instruction EX1 is 33 (decimal) or 011011 (binary). Similarly, the *opcode* value for EX2 is 64 (decimal), or 110100 (binary).

Figure 7.1 Sample Bit Encoding Table



7.1 Major Opcodes

Table 7.2 defines the major opcode for each instruction. The symbols used in the table are described in Table 7.1.

Every major opcode name starting with “POOL” requires a minor opcode, as defined in Section 7.2 “Minor Opcodes”. All other major opcodes refer to a particular instruction.

In the opcode tables, MSB denotes either bit 15 or 31, depending on instruction size.

Table 7.1 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).
∇	Operation or field codes marked with this symbol represent instructions which were only legal if 64-bit operations were enabled on implementations of Release 1 of the Architecture. In Release 2 of the architecture, operation or field codes marked with this symbol represent instructions which are legal if 64-bit floating point operations are enabled. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
Δ	Instructions formerly marked ∇ in some earlier versions of manuals, corrected and marked Δ in revision 5.03. Legal on MIPS64r1 but not MIPS32r1; in release 2 and above, legal in both MIPS64 and MIPS32, in particular even when running in “32-bit FPU Register File mode”, FR=0, as well as FR=1.
θ	Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, MIPS Technologies will assist the partner in selecting appropriate encodings if requested by the partner. The partner is not required to consult with MIPS Technologies when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (<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 Application-Specific Extensions. If the ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception.

Table 7.2 microMIPS64 Encoding of Major Opcode Field

Major		MSB..MSB-2							
MSB-3..MSB-5		0	1	2	3	4	5	6	7
		000	001	010	011	100	101	110	111
0	000	POOL32A δ	POOL32B δ	POOL32I δ	POOL32C δ	*	*	*	*
1	001	POOL16A δ	POOL16B δ	POOL16C δ	LWGP16	POOL16F	*	*	*
2	010	LBU16	LHU16	LWSP16	LW16	SB16	SH16	SWSP16	SW16
3	011	MOVE16	ANDI16	POOL16D δ	POOL16E δ	BEQZ16	BNEZ16	B16	LI16
4	100	ADDI32	ADDIU32	ORI32	XORI32	SLTI32	SLTIU32	ANDI32	JALX32
5	101	LBU32	LHU32	POOL32F δ	JALS32	BEQ32	BNE32	J32	JAL32
6	110	SB32	SH32	POOL32S δ	ADDIUPC	SWC132	SDC132	SD32 ⊥	SW32
7	111	LB32	LH32	DADDIU32 ⊥	*	LWC132	LDC132	LD32 ⊥	LW32

Examples:

1. The 32-bit instruction LW32 is assigned to the major opcode LW32 with the encoding “111111”.
2. The 16-bit instruction SUBU16 is assigned to the major opcode POOL16A with the encoding “000001”.

7.2 Minor Opcodes

While major opcodes have a fixed length of 6 bits, minor opcodes are variable in length. The minor opcodes are defined by opcode tables of one, two, or three dimensions, depending on the size of the opcode. Minor opcodes less than four bits are represented in a one-dimensional table (see [Table 7.15](#)), from four to six bits in a two-dimensional table (shown in [Figure 7.1](#) and [Table 7.9](#)), and from 7 to 10 bits in a three-dimensional table ([Table 7.4](#)). In a three-dimensional table, the two-dimensional table is expanded to include a column on the right side that encodes the extra bits. In the case of minor opcodes requiring multiple table cells, the instruction name appears in all cells, but the additional entries have a black background to indicate that this opcode is blocked (see [Table 7.4](#) and the legend shown in [Table 7.3](#)).

Example:

```
SRL r1, r1, 7    binary opcode fields: 000000 00001 00001 00111 00001 000000
interpretation:  POOL32A r1    r1    7        SRL
hex representation: 0021 3840
```

All minor opcode fields are right-aligned except those in 16-bit instructions and in 32-bit instructions with a 16-bit immediate field. These left-aligned fields are defined in a bit-reverse order, which is why, in order to accommodate the variable length of the field to the right, a given row and column in POOL32I represents bit 20..22 and 23..25 instead of bit 22..20 and 25..23.

If table entries are marked grey, then not all available bits of the instruction have been used for the encoding, leaving a field of empty bits. The empty bits are shown in the instruction tables in [Chapter 5, “microMIPS Re-encoded Instructions”](#) on page 51.

Table 7.3 Legend for Minor Opcode Tables

Symbol	Meaning
OPCODE	Occupied by Opcode
OPCODE	Space Utilized by another Opcode

Table 7.4 POOL32A Encoding of Minor Opcode Field

Minor	<i>bit 5..3</i>								<i>bit 2..0</i>	<i>bit 9..6</i>	
	0	1	2	3	4	5	6	7			
	000	001	010	011	100	101	110	111			
0	000	SLL32	*	SLLV	MOVN	*	*	*	*	0000	0
0	000	SRL32	*	SRLV	MOVZ	*	*	*	*	0001	1
0	000	SRA	*	SRAV	*	*	*	*	*	0010	2
0	000	ROTR	*	ROTRV	*	*	*	*	*	0011	3
0	000	*	*	ADD	LWXS	*	*	*	*	0100	4
0	000	*	*	ADDU32	*	*	*	*	*	0101	5
0	000	*	*	SUB	*	*	*	*	*	0110	6
0	000	*	*	SUBU32	*	*	*	*	*	0111	7
0	000	*	*	MUL	*	*	*	*	*	1000	8
0	000	*	*	AND	*	*	*	*	*	1001	9
0	000	*	*	OR32	*	*	*	*	*	1010	a
0	000	*	*	NOR	*	*	*	*	*	1011	b
0	000	*	*	XOR32	*	*	*	*	*	1100	c
0	000	*	*	SLT	*	*	*	*	*	1101	d
0	000	*	*	SLTU	*	*	*	*	*	1110	e
0	000	*	*	*	*	*	*	*	*	1111	f
1	001	SPECIAL2 θ	SPECIAL2 θ	SPECIAL2 θ	SPECIAL2 θ	SPECIAL2 θ	SPECIAL2 θ	SPECIAL2 θ	SPECIAL2 θ		*
2	010	COP2 θ	COP2 θ	COP2 θ	COP2 θ	COP2 θ	COP2 θ	COP2 θ	COP2 θ		
3	011	UDI θ	UDI θ	UDI θ	UDI θ	UDI θ	UDI θ	UDI θ	UDI θ		
4	100	*	INS	*	*	*	EXT	*		POOL32Axf δ	
5	101	ε	ε	ε	ε	ε	ε	ε	ε		
6	110	ε	ε								
7	111	BREAK32	*	*	*	ε	*	*	*		

Not Shown

SLL r0, r0, r0 = NOP
 SLL r0, r0, 1 = SSNOP
 SLL r0, r0, 3 = EHB
 SLL, r0, r0, 5 = PAUSE

Opcode Map

Table 7.5 POOL32Axf Encoding of Minor Opcode Extension Field

Extension		<i>bit 11..9</i>									
<i>bit 8..6</i>		0	1	2	3	4	5	6	7		
		000	001	010	011	100	101	110	111		
0	000	TEQ	TGE	TGEU	*	TLT	TLTU	TNE	*		
1	001	ε	ε	*	ε	ε	ε	*	ε		
2	010	ε	ε	ε	ε	ε	ε	ε	ε		
3	011	MFC0	MTC0	*	*	MFC0	MTC0				
<i>bit15..12</i>											
4	100	ε	ε	*	*	*	*	*	JALR / JR	0000	0
4	100	ε	ε	*	*	*	*	*	JALR.HB	0001	1
4	100	ε	*	*	*	*	SEB	*	*	0010	2
4	100	ε	*	*	*	*	SEH	*	*	0011	3
4	100	ε	*	*	*	*	CLO	MFC2	JALRS	0100	4
4	100	ε	*	*	*	*	CLZ	MTC2	JALRS.HB	0101	5
4	100	ε	*	*	*	*	RDHWR	DMFC2	*	0110	6
4	100	ε	ε	*	*	*	WSBH	DMTC2	*	0111	7
4	100		*	*	*	*	MULT	MFHC2	*	1000	8
4	100	ε	ε	*	*	*	MULTU	MTHC2	*	1001	9
4	100		*	*	*	*	DIV	*	*	1010	a
4	100	ε	ε	*	*	*	DIVU	*	*	1011	b
4	100	*	*	*	*	*	MADD	CFC2	*	1100	c
4	100	ε	ε	*	*	*	MADDU	CTC2	*	1101	d
4	100	*	*	*	*	*	MSUB	*	*	1110	e
4	100	ε	*	*	*	*	MSUBU	*	*	1111	f
<i>bit15..12</i>											
5	101	*	TLBP	ε	*	*	*	MFHI32	*	0000	0
5	101	*	TLBR	ε	*	*	*	MFLO32	*	0001	1
5	101	*	TLBWI	ε	*	*	*	MTHI	*	0010	2
5	101	*	TLBWR	ε	*	*	*	MTLO	*	0011	3
5	101	*	*	*	DI	*	*	*	*	0100	4
5	101	*	*	*	EI	*	*	*	*	0101	5
5	101	*	*	*	*	*	SYNC	*	*	0110	6
5	101	*	*	*	*	*	*	*	*	0111	7
5	101	*	*	*	*	*	SYSCALL	*	*	1000	8

Table 7.5 POOL32Axf Encoding of Minor Opcode Extension Field (Continued)

5	101	*	WAIT	*	*	*	*	*	*	1001	9
5	101	*	*	*	*	*	*	*	*	1010	a
5	101	*	*	*	*	*	*	*	*	1011	b
5	101	*	*	*	*	*	*	*	*	1100	c
5	101	*	ε	*	*	*	SDBBP	*	*	1101	d
5	101	RDPGPR	DERET	*	*	*	*	*	*	1110	e
5	101	WRPGPR	ERET	*	*	*	*	*	*	1111	f

6	110	ε	ε	*	*	ε	*	*	*
---	-----	---	---	---	---	---	---	---	---

7	111	ε	ε	ε	*	*	*	*	*
---	-----	---	---	---	---	---	---	---	---

Not Shown: JR = JALR r0

Table 7.6 POOL32F Encoding of Minor Opcode Field

Minor	bit 5..3							
	0	1	2	3	4	5	6	7
bit 2..0	000	001	010	011	100	101	110	111

bit 8..6

0	000	*	*	*	ε	MOV.fmt	*	ADD.fmt	MOVN.fmt	000	0
0	000	*	LWXC1 Δ	*	ε	MOV.T.fmt	*	SUB.fmt	MOVZ.fmt	001	1
0	000	PLL.PS ∇	SWXC1 Δ	*	ε	*	*	MUL.fmt	*	010	2
0	000	PLU.PS ∇	LDXC1 Δ	*	ε	*	*	DIV.fmt	*	011	3
0	000	PUL.PS ∇	SDXC1 Δ	*		*	*	ADD.fmt	MOVN.fmt	100	4
0	000	PUU.PS ∇	LUXC1 ∇	*		*	*	SUB.fmt	MOVZ.fmt	101	5
0	000	CVT.PS.S ∇	SUXC1 ∇	*	*	PREFX	*	MUL.fmt	*	110	6
0	000	*	*	*	*	*	*	DIV.fmt	*	111	7

1	001	MADD.S Δ	MADD.D Δ	MADD.PS ∇	ALNV.PS ∇	MSUB.S Δ	MSUB.D Δ	MSUB.PS ∇	*
---	-----	----------	----------	-----------	-----------	----------	----------	-----------	---

2	010	NMADD.S Δ	NMADD.D Δ	NMADD.PS ∇	*	NMSUB.S Δ	NMSUB.D Δ	NMSUB.PS ∇	*
---	-----	-----------	-----------	------------	---	-----------	-----------	------------	---

3	011	*	*	*	*	*	*	*	POOL32Fxf δ
---	-----	---	---	---	---	---	---	---	-------------

4	100	*	*	*	ε	*	*	*	C.cond.fmt
---	-----	---	---	---	---	---	---	---	------------

5	011	*	*	*	*	*	*	*	*
---	-----	---	---	---	---	---	---	---	---

6	100	*	*	*	*	*	*	*	*
---	-----	---	---	---	---	---	---	---	---

7	100	*	*	*	*	*	*	*	*
---	-----	---	---	---	---	---	---	---	---

Opcode Map

Table 7.7 POOL32Fxf Encoding of Minor Opcode Extension Field

Extension		<i>bit 10..8</i>									
		0	1	2	3	4	5	6	7		
<i>bit 7..6</i>		000	001	010	011	100	101	110	111		
											<i>bit 13..11</i>
0	00	*	CVT.L.fmt ▽	RSQRT.fmt Δ	FLOOR.L.fmt ▽	*	*	*	ε	000	0
0	00	*	CVT.W.fmt	SQRT.fmt	FLOOR.W.fmt	*	*	*	ε	001	1
0	00	CFC1	*	RECIP.fmt Δ	CEIL.L.fmt ▽	*	*	*	*	010	2
0	00	CTC1	*	*	CEIL.W.fmt	*	*	*	*	011	3
0	00	MFC1	CVT.S.PL ▽	*	TRUNC.L.fmt ▽	DMFC1	*		*	100	4
0	00	MTC1	CVT.S.PU ▽	*	TRUNC.W.fmt	DMTC1	*	*	*	101	5
0	00	MFHC1 ▽	*	*	ROUND.L.fmt ▽		*	*	*	110	6
0	00	MTHC1 ▽	*	*	ROUND.W.fmt	*	*	*	*	111	7
											<i>bit 12..11</i>
1	01	MOV.fmt	MOVF	*	ABS.fmt	*	*	*	ε	00	0
1	01	*	MOV _T	*	NEG.fmt	*	*	*	*	01	1
1	01	*	*	*	CVT.D.fmt	*	*	*	ε	10	2
1	01	*	*	*	CVT.S.fmt	*	*	*	*	11	3
*											
2	10	*	*	*	*	*	*	*	*		
*											
3	11	*	*	*	*	*	*	*	*		

Table 7.8 POOL32B Encoding of Minor Opcode Field

Minor		<i>bit 15</i>	
		0	1
<i>bit 14..12</i>		0	1
0	000	LWC2	SWC2
1	001	LWP	SWP
2	010	LDC2 ⊥	SDC2 ⊥
3	011	ε	ε
4	100	LDP ⊥	SDP ⊥
5	101	LWM32	SWM32
6	110	CACHE	*
7	111	LDM ⊥	SDM ⊥

Table 7.9 POOL32C Encoding of Minor Opcode Field

Minor		<i>bit 15</i>	
		0	1
<i>bit 14..12</i>		0	1
0	000	LWL	SWL
1	001	LWR	SWR
2	010	PREF	ST-EVA δ
3	011	LL	SC
4	100	LDL	SDL
5	101	LDR	SDR
6	110	LD-EVA δ	LWU
7	111	LLD	SCD

Table 7.10 LD-EVA Encoding of Minor Opcode Field

Minor		
<i>bit 11..9</i>		
0	000	LBUE
1	001	LHUE
2	010	LWLE
3	011	LWRE
4	100	LBE
5	101	LHE
6	110	LLE
7	111	LWE

Table 7.11 ST-EVA Encoding of Minor Opcode Field

Minor		
<i>bit 11..9</i>		
0	000	SWLE
1	001	SWRE
2	010	PREFE
3	011	CACHEE
4	100	SBE
5	101	SHE
6	110	SCE
7	111	SWE

Table 7.12 POOL32I Encoding of Minor Opcode Field

Minor		<i>bit 22..21</i>				
		0	1	2	3	
<i>bit 25..23</i>		00	01	10	11	
0	000	BLTZ	BLTZAL	BGEZ	BGEZAL	
1	001	BLEZ	BNEZC	BGTZ	BEQZC	
2	010	TLTI	TGEI	TLTIU	TGEIU	
3	011	TNEI	LUI	TEQI	*	
4	100	SYNCl	BLTZALS	*	BGEZALS	
5	101	BC2F	BC2T	*	*	
6	110	*	*	ε	ε	
<i>bit16</i>						
7	111	BC1F	BC1T	*	*	0
7	111	ε	ε	ε	ε	1

Table 7.13 POOL32S Encoding of Minor Opcode Field

Minor		<i>bit 5..3</i>									
		0	1	2	3	4	5	6	7		
<i>bit 2..0</i>		000	001	010	011	100	101	110	111		
<i>bit 8..6</i>											
0	000	DSLL	DSLL32	DSLLV	*	*	*	*	*	000	0
0	000	DSRL	DSRL32	DSRLV	*	*	*	*	*	001	1
0	000	DSRA	DSRA32	DSRAV	*	*	*	*	*	010	2
0	000	DROTR	DROTR32	DROTRV	*	*	*	*	*	011	3
0	000	*	*	DADD	*	*	*	*	*	100	4
0	000	*	*	DADDU	*	*	*	*	*	101	5
0	000	*	*	DSUB	*	*	*	*	*	110	6
0	000	*	*	DSUBU	*	*	*	*	*	111	7
1	001	ε	ε	ε	ε	ε	ε	ε	ε		

Table 7.13 POOL32S Encoding of Minor Opcode Field (Continued)

2	010	ε	ε	ε	ε	ε	ε	ε	ε
3	011	*	*	ε	ε	ε	ε	ε	ε
4	100	DINSM	DINS	DEXTU	DADDI	DEXTM	DEXT	DINSU	POOL32Sxf δ
5	101	ε	ε	ε	ε	ε	*	ε	ε
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Table 7.14 POOL32Sxf Encoding of Minor Opcode Extension Field

Extension		<i>bit 11..9</i>									
<i>bit 8..6</i>		0	1	2	3	4	5	6	7		
		000	001	010	011	100	101	110	111		
0	000	*	*	*	*	ε	ε	ε	ε		
1	001	*	ε	ε	ε	ε	ε	ε	ε		
2	010	ε	ε	ε	ε	ε	ε	ε	ε		
3	011	DMFC0	DMTC0	*	*	DMFC0	DMTC0	*	*		
<i>bit15..12</i>											
4	100	ε	ε	*	*	*	*	*	*	0000	0
4	100	ε	ε	*	*	*	*	*	*	0001	1
4	100	ε	ε	*	*	*	*	*	*	0010	2
4	100	*	*	*	*	*	*	*	*	0011	3
4	100	ε	*	*	*	*	DCLO	*	*	0100	4
4	100	ε	ε	ε	*	*	DCLZ	*	*	0101	5
4	100	ε	ε	ε	*	*	*	*	*	0110	6
4	100	ε	ε	*	*	*	DSBH	*	*	0111	7
4	100	*	*	*	*	*	DMULT	*	*	1000	8
4	100	ε	ε	*	*	*	DMULTU	*	*	1001	9
4	100	*	*	*	*	*	DDIV	*	*	1010	a

Table 7.14 POOL32Sxf Encoding of Minor Opcode Extension Field (Continued)

4	100	ε	ε	*	*	*	DDIVU	*	*	1011	b
4	100	*	*	*	*	*	*	*	*	1100	c
4	100	ε	ε	*	*	*	*	*	*	1101	d
4	100	*	*	*	*	*	*	*	*	1110	e
4	100	ε	*	*	*	*	DSHD	*	*	1111	f
5	101	*	*	*	*	*	*	*	*		
6	110	ε	ε	*	*	*	*	*	*		
7	111	ε	ε	ε	ε	*	*	*	*		

Table 7.15 POOL16A Encoding of Minor Opcode Field

Minor	
<i>bit 0</i>	
0	ADDU16
1	SUBU16

Table 7.16 POOL16B Encoding of Minor Opcode Field

Minor	
<i>bit 0</i>	
0	SLL16
1	SRL16

Table 7.17 POOL16C Encoding of Minor Opcode Field

Minor	<i>bit 6..4</i>								
	0	1	2	3	4	5	6	7	
<i>bit 9..7</i>	000	001	010	011	100	101	110	111	
0	000	NOT16	NOT16	NOT16	NOT16	XOR16	XOR16	XOR16	XOR16
1	001	AND16	AND16	AND16	AND16	OR16	OR16	OR16	OR16

Table 7.17 POOL16C Encoding of Minor Opcode Field

2	010	LWM16	LWM16	LWM16	LWM16	SWM16	SWM16	SWM16	SWM16
3	011	JR16	JR16	JRC	JRC	JALR16	JALR16	JALRS16	JALRS16
4	100	MFHI16	MFHI16	*	*	MFLO16	MFLO16	*	*
5	101	BREAK16	*	*	*	SDBBP16 σ	*	*	*
6	110	JRADDIUSP	JRADDIUSP	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Table 7.18 POOL16D Encoding of Minor Opcode Field

Minor	
<i>bit 0</i>	
0	ADDIUS5
1	ADDIUSP

Table 7.19 POOL16E Encoding of Minor Opcode Field

Minor	
<i>bit 0</i>	
0	ADDIUR2
1	ADDIUR1SP

Table 7.20 POOL16F Encoding of Minor Opcode Field

Minor	
<i>bit 0</i>	
0	MOVEP
1	*

7.3 Floating Point Unit Instruction Format Encodings

Instruction format encodings for the floating point unit are presented in this section.

If the instruction allows Single, Double and Pair-Single formats, the following encoding is used:

Table 7.21 Floating Point Unit Format Encodings - S, D, PS

<i>fmt</i> field		Mnemonic	Name	Bit Width	Data Type
Decimal	Hex				
0	0	S	Single	32	Floating Point
1	1	D	Double	64	Floating Point
2	2	PS	Paired Single	2 × 32	Floating Point
3	3	Reserved for future use by the architecture.			

If the instruction only allows Single and Double formats, the following encoding is used:

Table 7.22 Floating Point Unit Format Encodings - S, D 1-bit

<i>fmt</i> field		Mnemonic	Name	Bit Width	Data Type
Decimal	Hex				
0	0	S	Single	32	Floating Point
1	1	D	Double	64	Floating Point

Table 7.23 Floating Point Unit Instruction Format Encodings - S, D 2-bits

<i>fmt</i> field		Mnemonic	Name	Bit Width	Data Type
Decimal	Hex				
0	0	S	Single	32	Floating Point
1	1	D	Double	64	Floating Point
2, 3	2, 3	Reserved for future use by the architecture.			

If the instruction allows Single, Word and Long formats, the following encoding is used:

Table 7.24 Floating Point Unit Format Encodings - S, W, L

<i>fmt</i> field		Mnemonic	Name	Bit Width	Data Type
Decimal	Hex				
0	0	S	Single	32	Floating Point
1	1	W	Word	32	Integer
2	2	L	Long	64	Integer
3	3	Reserved for future use by the architecture.			

If the instruction allows Double, Word and Long formats, the following encoding is used:

Table 7.25 Floating Point Unit Format Encodings - D, W, L

<i>fmt</i> field		Mnemonic	Name	Bit Width	Data Type
Decimal	Hex				
0	0	D	Double	64	Floating Point
1	1	W	Word	32	Integer
2	2	L	Long	64	Integer
3	3	Reserved for future use by the architecture.			

Compatibility

This chapter covers various aspects of compatibility. microMIPS64 is the preferred replacement for the existing MIPS16e ASE and uses the same mode-switch mechanism. Although microMIPS includes almost all MIPS64 instructions and therefore does not require the original MIPS64 encodings, initially it will be implemented together with MIPS64-encoded instruction execution.

8.1 Assembly-Level Compatibility

microMIPS64 includes a re-encoding of the MIPS64 instructions, including all ASEs and UDI space. Therefore, microMIPS provides assembly-level compatibility. Only the following cases cause some side effects:

- **Re-encoded MIPS64 instructions with reduced operand fields**

There are 3 classes of reduced fields:

1. *Reserved or unsupported bits and encodings.* This category is not a problem because utilizing a reserved or unsupported field causes an exception, no operation, or undefined behavior, and often these cannot be accessed by the compiler anyway. An example of this category is the ‘fmt’ field.
2. *Bit fields and ranges which are defined but typically never used.* This category is usually not a problem. The assembler generates an error message if a constant is outside of the re-defined range.
3. *Bit fields which are used but were reduced in order to utilize the new opcode map most efficiently.* The handling of these cases is similar to category 2 above—compilers do not generate such scenarios, and assemblers generate error messages. In the latter case, the programmer has to either fix the code or switch to the MIPS64 encoding.

- **Re-encoded Branch and Jump instructions**

Branch instructions support 16-bit aligned branch target addresses, providing full flexibility for microMIPS. Because the offset field size of the 32-bit encoded branch instructions is the same as the MIPS64-encoded instructions, and because all branch target addresses of the MIPS64 encoding are 32-bit aligned, the branch range in microMIPS is smaller. This is partially compensated by the smaller code size of microMIPS.

Jump instructions also support 16-bit aligned target addresses. This reduces the addressable target region for J, JAL to 128 MB instead of 256 MB. For these instructions, the effective target address is in the ‘current’ 128 MB-aligned region. For larger ranges, the jump register instructions (JR, JRC, and JRADDIUSP) can be used.

- **MIPS64 assembly instructions manually encoded using the .WORD directive**

Manual encoding of MIPS64 assembly instructions can be used in assembly code as well as assembly macros in C functions. To differentiate between microMIPS-encoded instructions and other encoded instructions or data, the following compiler directives have been introduced:

```
.set micromips    ; instruction stream is microMIPS

.set nomicromips ; instruction stream is MIPS64

.insn            ; If in microMIPS instruction stream mode, the location associated
                ; with the previous label is aligned to 16-bit bits instead of
                ; 32-bits
                ; If in microMIPS instruction stream mode and if the previous
                ; label is loaded to a register as the target of a jump or branch,
                ; the ISAMode bit is set within the branch/jump register value.
```

The programmer must use these directives to encode instructions in microMIPS.

For example, to manually encode a microMIPS NOP:

```
.set micromips

label1:
    .insn
    .word 0      ; label1 location - represents microMIPS NOP32 instruction
label2:
    .insn
    .half 0x0c00 ; label2 location - represents microMIPS NOP16 instruction
label3:
    .half 0x0c00 ; label3 location - represents data value of 3072 (decimal)
```

To manually encode a MIPS64 NOP:

```
.set nomicromips
.word 0      ; represents MIPS64 NOP instruction
```

For MIPS64 instruction stream mode, the “.insn” directive has no effect.

- **Branch likely instructions**

microMIPS does not support branch likely instructions in hardware. Assembly-level compatibility is maintained because assemblers replace branch likely instructions either by an instruction sequence or by a regular branch instruction, and they perform some instruction reordering if reordering is possible.

8.2 ABI Compatibility

microMIPS is compatible with the existing ABIs o32, n32, and n64 calling conventions. However, a few new relocation types need to be added to these ABIs for microMIPS support, as some of the additional offset field sizes required for microMIPS become visible to the linker. For example, the offset fields of J and SW using GP are visible to the linker, while B and SWSP are hidden within the object files.

Functions remain 32-bit aligned as in the MIPS64 encoding as well as MIPS16e. This guarantees that static and dynamic linking processes can link microMIPS object files with MIPS64 object files.

Compatibility

Programs can be composed of both microMIPS and MIPS64 modules, using either the JALX instructions (and/or JR instructions with setting the ISAMode bit appropriately) to switch instruction set modes when calling routines compiled in an ISA different from that of the caller routine.

microMIPS provides flexibility for potential future ABIs.

8.3 Branch and Jump Offsets

microMIPS branch targets are half-word (16-bit) aligned to match half-word sized instructions. Please refer to [Section 3.6, "Branch and Jump Offsets."](#)

8.4 Relocation Types

Compiler and linker toolchains need to be modified with new relocation types to support microMIPS. Reasons for these new relocation types include:

1. The placement of instruction halfwords is determined by memory endianness. MIPS64 instructions are always of word size, so there were no halfword placement issues.
2. microMIPS has 7-bit, 10-bit and 16-bit PC-relative offsets.
3. Branch and Jump offset fields are left-shifted by 1 bit (instead of 2 bits in MIPS64) to create effective target addresses.
4. Some code-size optimizations can only be done at link time instead of compile time. Some new relocation types are used solely within the linker to keep track of address and data information.

8.5 Boot-up Code shared between microMIPS64 and MIPS64

In some systems, it would be advantageous to place both microMIPS64 and MIPS64 executables in the same boot memory. In that way, a single system could be used for either instruction set.

To enable this, a binary code sequence is required that can be run in either instruction set and change code paths depending on the instruction set that is being used.

The following binary sequence achieves this goal:

```
0x1000wxyz // where w,x,y,z represent hexadecimal digits
0x00000000
```

For the MIPS64 instruction set, this binary sequence is interpreted as:

```
BEQ $0, $0, wxyz // branch to location of more MIPS64 instructions
NOP
```

For the microMIPS instruction set, this binary sequence is interpreted as:

```
ADDI32 $0, $0, wxyz // do nothing
NOP // fall through to more microMIPS instructions
```

8.6 Coprocessor Unusable Behavior

When a coprocessor instruction is executed when the associated coprocessor has not been implemented, it is allowed for the RI exception to be signalled instead of the Coprocessor Unusable exception. Please refer to [Section 3.7](#), "Coprocessor Unusable Behavior."

8.7 Other Issues Affecting Software and Compatibility

microMIPS instructions can cross cache lines and page boundaries. Hardware must handle these cases so that software need not avoid them. Since MIPS64 requires instructions to be 32-bit aligned, there is no forward compatibility issue when transitioning to microMIPS.

References

This appendix lists other publications available from MIPS Technologies, some of which are referenced elsewhere in this document. They may be included in the `$MIPS_HOME/$MIPS_CORE/doc` area of a typical soft or hard core release, or in some cases may be available on the MIPS web site, <http://www.mips.com>.

1. MIPS® Architecture For Programmers, Volume I: Introduction to the MIPS64® Architecture
MIPS document: MD0083
2. MIPS® Architecture For Programmers, Volume II: The MIPS64® Instruction Set
MIPS document: MD0087
3. MIPS® Architecture For Programmers, Volume III: The MIPS64® and microMIPS64™ Privileged Resource Architecture
MIPS document: MD0087

Revision History

Revision	Date	Description
1.08	November 25, 2009	<ul style="list-style-type: none"> • Clean-up for external release.
1.09	January 7, 2010	<ul style="list-style-type: none"> • Added shared boot-up code sequence in Compatibility Chapter.
3.00	March 25, 2010	<ul style="list-style-type: none"> • Changed document revision numbering to match other Release 3 documents. Hopefully this will be less confusing. • Moved MIPS32/64 version of JALX to Volume II-A.
3.01	October 30, 2010	<ul style="list-style-type: none"> • User mode instructions not allowed to product UNDEFINED results. • Updated copyright page. • Removed Margin Note - “Preliminary - Subject to Change” in some chapters.
3.02	December 6, 2010	<ul style="list-style-type: none"> • POOL32Sxf binary encoding was incorrect for individual instruction description pages.
3.03	December 10, 2010	<ul style="list-style-type: none"> • microMIPS AFP versions security reclassification.
3.04	March 21, 2011	<ul style="list-style-type: none"> • RSQRT/RECIP does not need 64-bit FPU. • MADD fmt/NMADD.fmt/MSUB fmt/NMSUB.fmt psuedo-code was incorrect for PS format check.
3.05	April 4, 2011	<ul style="list-style-type: none"> • The text description was incorrect for the offset sizes for these instructions - CACHE, LDC2, LL, LWC2, LWL, LWR, PREF, SDC2, SWL, SWR. • CACHE & WAIT instruction descriptions were using the wrong instruction bit numbers. • LWU was incorrectly included int the microMIPS32 version.
3.06	October 17, 2012	<ul style="list-style-type: none"> • CVT.D fmt and CVT.S fmt were in wrong positions within Table POOL32Fxf.
3.07	October 26, 2012	<ul style="list-style-type: none"> • Fix Figure 6.1 - columns & rows were transposed from the real tables.
5.00	December 14, 2012	<ul style="list-style-type: none"> • Some of the microMIPS instructions were not listed in alphabetical order. Fixed. No content change. • R5 changes: DSP and MT ASEs -> Modules • NMADD fmt, NMSUB fmt - for IEEE2008 negate portion is arithmetic.
5.01	December 16, 2012	<ul style="list-style-type: none"> • No technical context change: • Update cover with microMIPS logo • Update copyright text. • Update pdf filename.

Revision	Date	Description
5.03	August 21, 2012	<ul style="list-style-type: none"> Resolved inconsistencies with regards to the availability of instructions in MIPS32r2: MADD fmt family (MADD.S, MADD.D, NMADD.S, NMADD.D, MSUB.S, MSUB.D, NMSUB.S, NMSUB.D), RECIP fmt family (RECIP.S, RECIP.D, RSQRT.S, RSQRT.D), and indexed FP loads and stores (LWXC1, LDXC1, SWXC1, SDXC1). These instructions are required to be available in all FPUs. .
5.04	January 15, 2014	<p>LLSC Related Changes</p> <ul style="list-style-type: none"> Added ERETNC. New. Modified SC handling: refined, added, and elaborated cases where SC can fail or was UNPREDICTABLE. <p>XPA Related Changes</p> <ul style="list-style-type: none"> Added MTHC0, MFHC0 to access extensions. All new. Modified MTC0 for MIPS32 to zero out the extended bits which are writeable. This is to support compatibility of XPA hardware with non XPA software. In pseudo-code, added registers that are impacted. MTHC0 and MFHC0 - Added RI conditions.

Revision History