MIPS® Architecture for Programmers Volume II-B: microMIPS32™ Instruction Set

Document Number: MD00582
Revision 6.01
June 9, 2015

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Public. This publication contains proprietary information which is subject to change without notice and is supplied ‘as is’, without any warranty of any kind.

Template: nB1.03, Built with tags: 2B ARCH

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Contents

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Chapter 1

About This Book

The MIPS® Architecture for Programmers Volume II-B: microMIPS32™ 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 MIPS32® Architecture
- Volume I-B describes conventions used throughout the document set, and provides an introduction to the microMIPS™ Architecture
- Volume II-A provides detailed descriptions of each instruction in the MIPS32® instruction set
- Volume II-B provides detailed descriptions of each instruction in the microMIPS32™ instruction set
- Volume III describes the MIPS32® and microMIPS32™ 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 MIPS32® Architecture. Beginning with Release 3 of the Architecture, microMIPS is the preferred solution for smaller code size. Release 6 removes MIPS16e: MIPS16e cannot be implemented with Release 6.
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64® Architecture and microMIPS64™. With Release 5 of the Architecture, MDMX is deprecated. MDMX and MSA can not be implemented at the same time. Release 6 removes MDMX: MDMX cannot be implemented with Release 6.
- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS® Architecture. Release 6 removes MIPS-3D: MIPS-3D cannot be implemented with Release 6.
- Volume IV-d describes the SmartMIPS®Application-Specific Extension to the MIPS32® Architecture and the microMIPS32™ Architecture . Release 6 removes SmartMIPS: SmartMIPS cannot be implemented with Release 6, neither MIPS32 Release 6 nor MIPS64 Release 6.
- 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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

1.1 Typographical Conventions

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

1.1.1 Italic Text

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

1.1.2 Bold Text

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

1.1.3 Courier Text

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

1.2 UNPREDICTABLE and UNDEFINED

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

1.2.1 UNPREDICTABLE

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

UNPREDICTABLE results or operations have several implementation restrictions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

- **UNPREDICTABLE** operations must not read, write, or modify the contents of memory or internal state which is inaccessible in the current processor mode. For example, **UNPREDICTABLE** operations executed in user mode must not access memory or internal state that is only accessible in Kernel Mode or Debug Mode or in another process

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

1.2.2 UNDEFINED

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

UNDEFINED operations or behavior has one implementation restriction:

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

1.2.3 UNSTABLE

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

UNSTABLE values have one implementation restriction:

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

1.3 Special Symbols in Pseudocode Notation

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

Table 1.1 Symbols Used in Instruction Operation Statements

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
CPR[z,x,s]	Coprocessor unit *z*, general register *x*, select *s*
CP2CPR[x]	Coprocessor unit 2, general register *x*
CCR[z,x]	Coprocessor unit *z*, control register *x*
CP2CCR[x]	Coprocessor unit 2, control register *x*
COC[z]	Coprocessor unit *z* condition signal
Xlat[x]	Translation of the MIPS16e GPR number *x* into the corresponding 32-bit GPR number
BigEndianMem	Endian mode as configured at chip reset (0 → Little-Endian, 1 → Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions) and the endianness of Kernel and Supervisor mode execution.
BigEndianCPU	The endianness for load and store instructions (0 → Little-Endian, 1 → Big-Endian). In User mode, this endianness may be switched by setting the *RE* bit in the *Status* register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).
ReverseEndian	Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as (SR_{RE} and User mode).
LLbit	Bit of **virtual** state used to specify operation for instructions that provide atomic read-modify-write. *LLbit* is set when a linked load occurs and is tested by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.
I:, **I+n**:, **I-n**:	This occurs as a prefix to *Operation* description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to "execute." Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of **I**. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction **I**, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled **I+1**. The effect of pseudocode statements for the current instruction labeled **I+1** appears to occur "at the same time" as the effect of pseudocode statements labeled **I** for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur "at the same time," there is no defined order. Programs must not depend on a particular order of evaluation between such sections.
PC	The *Program Counter* value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to *PC* during an instruction time. If no value is assigned to *PC* during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the *PC* during the instruction time of the instruction in the branch delay slot. In the MIPS Architecture, the PC value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. Release 6 adds PC-relative address computation and load instructions. The PC value contains a full -bit address, all of which are significant during a memory reference.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

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

1.4 Notation for Register Field Accessibility

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

Table 1.2 Read/Write Register Field Notation

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 1.2 Read/Write Register Field Notation (Continued)

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 1.2 Read/Write Register Field Notation (Continued)

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

1.5 For More Information

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

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Chapter 2

Introduction

In today's market, the lowest price, performance, or both 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 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 MIPS® instruction encoding and can be implemented in parallel or standalone. The microMIPS equivalent of MIPS32 is microMIPS32™ and the microMIPS equivalent of MIPS64 is microMIPS64™.

Overview of changes vs. existing MIPS ISA:

- 16-bit and 32-bit opcodes
- Optimized opcode/operand field definitions based on statistics
- Removal of branch likely instructions, emulation by assembler
- Fine-tuned register allocation algorithm in the compiler for lowest code size

2.1 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 microMIPS.

For implementations that support both microMIPS and MIPS, 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 microMIPS and MIPS, 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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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 microMIPS and MIPS, 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.

2.2 Software Detection

Software can determine if microMIPS 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 microMIPS.

Software can determine if the MIPS 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.

2.3 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.

2.4 ISA Mode Switch

The MIPS Release 3 architecture defines an ISA mode for each processor. An ISA mode value of 0 indicates MIPS instruction decoding. In processors implementing microMIPS, an ISA mode value of 1 selects microMIPS instruction decoding.

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 MIPS ISA, this means the JALR and JALR.HB instructions. For the microMIPS ISA, this means the JALRC, JALRC.HB, JIALC, and JALRC16 instructions.

Jump-Register instructions: For the MIPS ISA, this means the JR and JR.HB instructions. For the microMIPS ISA, this means the instructions JRC, JRC.HB, JIC, JRC16, and JRCADDIUSP instructions.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Mode switching between MIPS and microMIPS is enabled by the Jump-and-Link-Register and Jump-Register instructions, as described below.

- The Jump-and-Link-Register and Jump-Register instructions interpret bit 0 of the source registers as the target ISA mode (0=MIPS, 1=microMIPS) 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 $Config3_{ISAOnExc}$ 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 MIPS or microMIPS) then this mode switch mechanism does not exist, but the ISA Mode bit is still maintained and has a fixed value (0=MIPS, 1=microMIPS). This is to maintain code compatibility between devices which implement both ISA modes and devices which implement only one ISA mode. 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.

2.5 Branch and Jump Offsets

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

2.6 Coprocessor Unusable Behavior

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

If the microMIPS implementation chooses to use RI exception in such cases, the microMIPS RI exception handler must check for coprocessor instructions being executed while the associated coprocessor is implemented but has been disabled ($Status_{CUx}$ set to zero).

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

2.7 Release 6 of the MIPS Architecture

Table 2.1 Instructions Added in Release 6

Instruction	Instruction's Purpose	Replaces
ADDIUPC	Add Immediate to PC (unsigned - non-trapping)	New
ALIGN	Concatenate two GPRs, and extract a contiguous subset at a byte position (32-bit)	New
ALUIPC	Aligned Add Upper Immediate to PC	New
AUI	Add Upper Immediate	New
AUIPC	Add Upper Immediate to PC	New
BC1EQZC	Branch if Coprocessor 1 (FPU) Register Bit 0 is Equal to Zero	BCIF
BC1NEZC	Branch if Coprocessor 1 (FPR) Register Bit 0 is Not Equal to Zero	BCIT
BC2EQZC	Branch if Coprocessor 2 Condition Register is Equal to Zero	BC2F
BC2NEZC	Branch if Coprocessor 2 Condition Register is Not Equal to Zero	BC2T
BLEZALC	Compact branch-and-link if GPR rt is less than or equal to zero	New
BGEZALC	Compact branch-and-link if GPR rt is greater than or equal to zero	Compact version
BGTZALC	Compact branch-and-link if GPR rt is greater than zero	New
BLTZALC	Compact branch-and-link if GPR rt is less than to zero	Compact version
BEQZALC	Compact branch-and-link if GPR rt is equal to zero	New
BNEZALC	Compact branch-and-link if GPR rt is not equal to zero	New
BEQC	Equal register-register compare and branch with 16-bit offset	New
BNEC	Not-Equal register-register compare and branch with 16-bit offset	New
BLTC	Signed register-register compare and branch with 16-bit offset:67	New
BGEC	Signed register-register compare and branch with 16-bit offset:	New
BLTUC	Unsigned register-register compare and branch with 16-bit offset:	New
BGEUC	Unsigned register-register compare and branch with 16-bit offset:	New
BGTC	Assembly idioms with reversed operands for signed/unsigned compare-and-branch	New
BLEC	Assembly idioms with reversed operands for signed/unsigned compare-and-branch	New
BGTUC	Assembly idioms with reversed operands for signed/unsigned compare-and-branch	New
BLEUC	Assembly idioms with reversed operands for signed/unsigned compare-and-branch	New
BLTZC	Signed Compare register to Zero and branch with 16-bit offset	Compact version
BLEZC	Signed Compare register to Zero and branch with 16-bit offset	Compact version
BGEZC	Signed Compare register to Zero and branch with 16-bit offset	Compact version
BGTZC	Signed Compare register to Zero and branch with 16-bit offset	Compact version
BEQZC	Equal Compare register to Zero and branch with 21-bit offset	Compact version with 21-bit offset

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 2.1 Instructions Added in Release 6 (Continued)

Instruction	Instruction's Purpose	Replaces
BNEZC	Not-equal Compare register to Zero and branch with 21-bit offset	Compact version with 21-bit offset
BC/BC16	Branch, Compact (16)	B/B16
BALC	Branch and Link, Compact	BAL
BITSWAP	Swaps (reverses) bits in each byte	New
BOVC	Branch on Overflow, Compact; Branch on No Overflow, Compact	New
BNVC	Branch on Overflow, Compact; Branch on No Overflow, Compact	New
CLASS.fmt	Scalar Floating-Point Class Mask	New
CMP.condn.fmt	Floating Point Compare setting Mask	C.condn.fmt
DIV	Divide Words Signed	DIV
DVP	Disable Virtual Processor	New
EVP	Enable Virtual Processor	New
MOD	Modulo Words Signed	DIV
DIVU	Divide Words Signed	DIVU
MODU	Modulo Words Signed	DIVU
JALRC16	Jump and Link Register Compact (16-bit instr size)	JALR16
JIALC	Jump Indexed and Link, Compact	New
JIC	Jump Indexed, Compact	New
JRCADDIUSP	Jump Register, Adjust Stack Pointer (16-bit)	JRADDIUSP
LDPC	Load Doubleword PC-relative	New
LSA	Load Scaled Address	New
MADDF.fmt	Floating Point Fused Multiply Add	MADD.fmt
MSUBF.fmt	Floating Point Fused Multiply Subtract	MSUB.fmt
MAX.fmt	Scalar Floating-Point Maximum	New
MAXA.fmt	Scalar Floating-Point Argument with Maximum Absolute Value	New
MIN.fmt	Scalar Floating-Point Minimum	New
MINA.fmt	Scalar Floating-Point Argument with Minimum Absolute Value	New
MUL	Multiply Words Signed, Low Word	MULT
MUH	Multiply Words Signed, High Word	MULT
MULU	Multiply Words Signed, Low Word	MULTU
MUHU	Multiply Words Signed, High Word	MULTU
NAL	No-op and Link	Retained from BLTZAL
RINT.fmt	Floating-Point Round to Integral	New
SEL.fmt	Select floating point values with FPR condition	MOVF.fmt, MOVT.fmt
SELEQZ	Select integer GPR value or zero	MOVZ, MOVN
SELNEZ	Select integer GPR value or zero	MOVZ, MOVN
SELEQZ.fmt	Select floating point value or zero with FPR condition	MOVZ.fmt, MOVN.fmt
SELNEZ.fmt	Select floating point value or zero with FPR condition	MOVZ.fmt, MOVN.fmt

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 2.2 Instructions Recoded in Release 6

Instruction	Purpose
AND16	To do a bitwise logical AND
BEQZC	Branch on Equal to Zero, Compact
BNEZC	Branch on Not Equal to Zero, Compact
BREAK16	Breakpoint
JRC16	Jump Register, Compact (16-bit)
LUI	To load a constant into the upper half of a word
LWM16	Load Word Multiple (16-bit)
MOVEP	Move a Pair of Registers
NOT16	Invert (16-bit instr size)
OR16	Or (16-bit instr size)
SDBBP16	Software Debug Breakpoint (16-bit instr size)
SWM16	Store Word Multiple (16-bit)
SYNCI	Synchronize Caches to Make Instruction Writes Effective
XOR16	Exclusive OR (16-bit instr size)

Table 2.3 Instructions Removed in Release 6

Instruction	Purpose	Replaced by
ABS.PS	Floating Point Absolute Value, Paired Single	—
ADD.PS	Floating Point Add, Paired Single	—
ADDI	Add Immediate Word	—
ALNV.PS	Floating Point Align Variable, Paired Single	—
B	Unconditional Branch	BC
B16	Unconditional Branch (16-bit instr size)	BC16
BAL	Branch and Link	BALC
BC1F	Branch on FP False	BC1EQZC
BC1T	Branch on FP True	BC1NEZC
BC2F	Branch on COP2 False	BC2EQZC
BC2T	Branch on COP2 True	BC2NEZC
BEQ	Branch on Equal	BEQC

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 2.3 Instructions Removed in Release 6 (Continued)

Instruction	Purpose	Replaced by
BGEZ	Branch on Greater Than or Equal to Zero	BGEZC
BEQZ16	Branch on Equal to Zero (16-bit instr size)	BEQZC16
BGEZAL	Branch on Greater Than or Equal to Zero and Link	BGEZALC
BGEZALS	Branch on Greater Than or Equal to Zero and Link, Short Delay-Slot	—
BGTZ	Branch on Greater Than Zero	BGTZC
BLEZ	Branch on Less Than or Equal to Zero	BLEZC
BLTZ	Branch on Less Than Zero	BLTZC
BLTZAL	Branch on Less Than Zero and Link	BLTZALC
BLTZALS	Branch on Less Than Zero and Link, Short Delay-Slot	BLTZALC.
BNE	Branch on Not Equal	BNEC
BNEZ16	Branch on Not Equal to Zero (16-bit instr size)	BNEZC16.
C.cond.fmt	Floating Point Compare	CMP.condn.fmt
CVT.PS.S	Floating Point Convert Pair to Paired Single	—
CVT.S.PL	Floating Point Convert Pair Lower to Single Floating Point	—
CVT.S.PU	Floating Point Convert Pair Upper to Single Floating Point	—
DADDI	Doubleword Add Immediate	—
DIV	Divide Word	—
DIVU	Divide Unsigned Word	—
DMULT	Doubleword Multiply	—
DMULTU	Doubleword Multiply Unsigned	—
JALC	Jump and Link Compact	—
JALR16	Jump and Link Register (16-bit instr size)	JALRC16
JALRS	Jump and Link Register, Short Delay Slot	JALRC
JALRS.HB	Jump and Link Register with Hazard Barrier, Short Delay-Slot	—
JALRS16	Jump and Link Register, Short Delay-Slot (16-bit instr size)	JALRC16
JALS	Jump and Link, Short Delay Slot	—
JALX	Jump and Link Exchange (microMIPS Format)	—
JC	Jump Register, Compact	—
JR	Jump Register	JALRC
JR.HB	Jump Register with Hazard Barrier	JALRC.HB
JRC	Jump Register, Compact (16)	—

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 2.3 Instructions Removed in Release 6 (Continued)

Instruction	Purpose	Replaced by
JR16	Jump Register (16-bit instr size)	JRC16
JRADDIUSP	Jump Register, Adjust Stack Pointer	JRCADDIUSP
LDL	Load Doubleword Left	—
LDR	Load Doubleword Right	—
LDXC1	Load Doubleword Indexed to Floating Point	—
LUXC1	Load Doubleword Indexed Unaligned to Floating Point	—
LWL	Load Word Left	—
LWLE	Load Word Left EVA	—
LWR	Load Word Right	—
LWRE	Load Word Right EVA	—
LWXC1	Load Word Indexed to Floating Point	—
LWXS	Load Word Indexed, Sealed	—
MADD	Multiply and Add Word to Hi, Lo	—
MADD.fmt	Floating Point Multiply Add	MADDF.fmt
MADDU	Multiply and Add Unsigned Word to Hi,Lo	—
MFHI16	Move From HI Register (16-bit instr size)	—
MFLO16	Move From LO Register	—
MFHI	Move From HI Register	—
MFLO	Move From LO Register	—
MOV.PS	Floating Point Move	—
MOVF.fmt	Floating Point Move Conditional on Floating Point False	SEL.fmt
MOVN	Move Conditional on Not Zero	SELNEZ, SELEQZ
MOVN.fmt	Floating Point Move Conditional on Not Zero	SELNEZ.fmt
MOVT	Move Conditional on Floating Point True	—
MOVT.fmt	Floating Point Move Conditional on Floating Point True	SEL.fmt
MOVZ	Move Conditional on Zero	SELNEZ, SELEQZ
MOVZ.fmt	Floating Point Move Conditional on Zero	SELEZQZ.fmt
MSUB	Multiply and Subtract Word to Hi, Lo	—
MSUB.fmt	Floating Point Multiply Subtract	MSUBF.fmt
MSUBU	Multiply and Subtract Word to Hi,Lo	—
MTHI	Move to HI Register	—

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 2.3 Instructions Removed in Release 6 (Continued)

Instruction	Purpose	Replaced by
MTLO	Move to LO Register	—
MUL	Multiply Word to GPR	—
MUL.PS	Floating Point Multiply, Paired Single	—
MULT	Multiply Word	MUL, MULH
MULTU	Multiply Unsigned Word	MULU, MUHU
NEG.PS	Floating Point Negate, Paired Single	—
NMADD.fmt	Floating Point Negative Multiply Add	NMADDF.fmt
NMSUB.fmt	Floating Point Negative Multiply Subtract	NMSUBF.fmt
PLL.PS	Pair Lower Lower, Paired Single	—
PLU.PS	Pair Lower Upper, Paired Single	—
PREFX	Prefetch Indexed	—
PUL.PS	Pair Upper Lower, Paired Single	—
PUU.PS	Pair Upper Upper, Paired Single	—
SDL	Store Doubleword Left	—
SDR	Store Doubleword Right	—
SDXC1	Store Doubleword Indexed from Floating Point	—
SUB.PS	Floating Point Subtract	—
SUXC1	Store Doubleword Indexed Unaligned from Floating Point	—
SWL	Store Word Left	—
SWLE	Store Word Left EVA	—
SWR	Store Word Right	—
SWXC1	Store Word Indexed from Floating Point	—
TEQI	Trap if Equal Immediate	—
TGEI	Trap if Greater or Equal Immediate	—
TGEIU	Trap if Greater or Equal Immediate Unsigned	—
TLTI	Trap if Less Than Immediate	—
TLTIU	Trap if Less Than Immediate Unsigned	—
TNEI	Trap if Not Equal Immediate	—

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Chapter 3

Guide to the Instruction Set

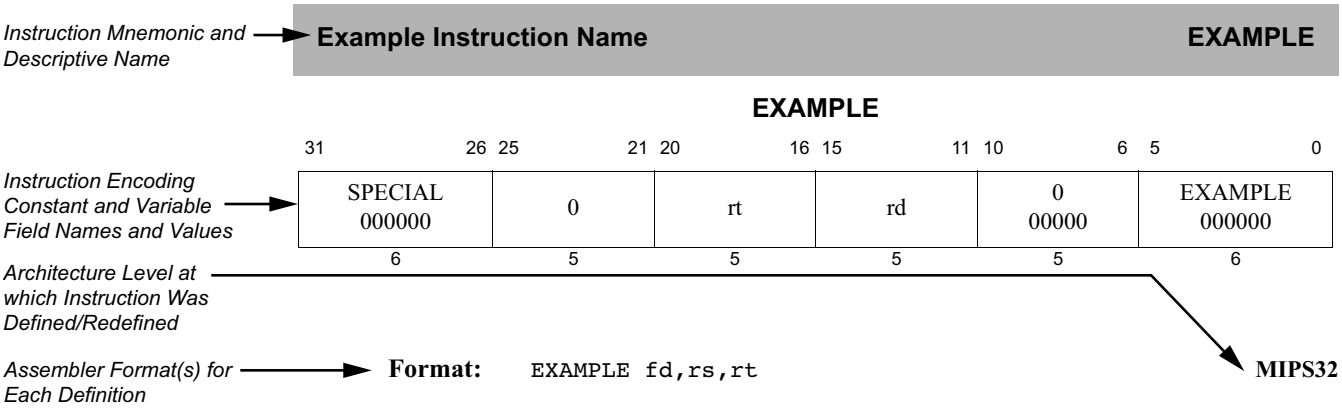
This chapter provides a detailed guide to understanding the instruction descriptions, which are listed in alphabetical order in the tables at the beginning of the next chapter.

3.1 Understanding the Instruction Fields

Figure 3.1 shows an example instruction. Following the figure are descriptions of the fields listed below:

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Figure 3.1 Example of Instruction Description



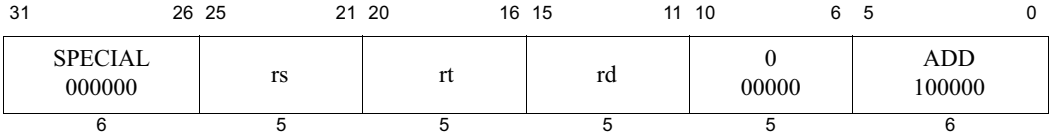
Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

3.1.1 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 3.2). 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 3.2).
- Fields that contain zeros but are not named are unused fields that are required to be zero (bits 10:6 in Figure 3.2). If such fields are set to non-zero values, the operation of the processor is **UNPREDICTABLE**.

Figure 3.2 Example of Instruction Fields



3.1.2 Instruction Descriptive Name and Mnemonic

The instruction descriptive name and mnemonic are printed as page headings for each instruction, as shown in Figure 3.3.

Figure 3.3 Example of Instruction Descriptive Name and Mnemonic

Add Word **ADD**

3.1.3 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 3.4 Example of Instruction Format

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

The assembler format is shown with literal parts of the assembler instruction printed in uppercase characters. The variable parts, the operands, are shown as the lowercase names of the appropriate fields.

The architectural level at which the instruction was first defined, for example “MIPS32” is shown at the right side of the page. Instructions introduced at different times by different ISA family members, are indicated by markings such

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

as “MIPS64, MIPS32 Release 2”. Instructions removed by particular architecture release are indicated in the Availability section.

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

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

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.

3.1.4 Purpose Field

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

Figure 3.5 Example of Instruction Purpose

Purpose: Add Word

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

3.1.5 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 3.6 Example of Instruction Description

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

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

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

3.1.6 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)
- 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 3.7 Example of Instruction Restrictions

Restrictions:

3.1.7 Availability and Compatibility Fields

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

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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

3.1.8 Operation Field

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

Figure 3.8 Example of Instruction Operation

Operation:

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

See 3.2 “Operation Section Notation and Functions” on page 33 for more information on the formal notation used here.

3.1.9 Exceptions Field

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

Figure 3.9 Example of Instruction Exception

Exceptions:

Integer Overflow

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

3.1.10 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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Figure 3.10 Example of Instruction Programming Notes

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

3.2 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:

3.2.1 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).

3.2.2 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:

3.2.2.1 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.

3.2.2.1.1 COP_LW

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

Figure 3.11 COP_LW Pseudocode Function

```
COP_LW (z, rt, memword)
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
    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
```

3.2.2.1.2 COP_LD

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

Figure 3.12 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
```

3.2.2.1.3 COP_SW

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

Figure 3.13 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
```

3.2.2.1.4 COP_SD

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

Figure 3.14 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 */
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
endfunction COP_SD
```

3.2.2.1.5 CoprocessorOperation

The CoprocessorOperation function performs the specified Coprocessor operation.

Figure 3.15 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
```

3.2.2.2 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 3.1. 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.

3.2.2.2.1 Misaligned Support

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

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

Figure 3.16 MisalignedSupport Pseudocode Function

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

See Appendix B, “Misaligned Memory Accesses” on page 511 for a more detailed discussion of misalignment, including pseudocode functions for the actual misaligned memory access.

3.2.2.2.2 AddressTranslation

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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 3.17 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
```

3.2.2.2.3 LoadMemory

The LoadMemory function loads a value from memory.

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

Figure 3.18 LoadMemory Pseudocode Function

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

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

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

endfunction LoadMemory
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

3.2.2.2.4 StoreMemory

The StoreMemory function stores a value to memory.

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

Figure 3.19 StoreMemory Pseudocode Function

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

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

endfunction StoreMemory
```

3.2.2.2.5 Prefetch

The Prefetch function prefetches data from memory.

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

Figure 3.20 Prefetch Pseudocode Function

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

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

endfunction Prefetch
```

Table 3.1 lists the data access lengths and their labels for loads and stores.

Table 3.1 AccessLength Specifications for Loads/Stores

AccessLength Name	Value	Meaning
DOUBLEWORD	7	8 bytes (64 bits)

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 3.1 AccessLength Specifications for Loads/Stores

AccessLength Name	Value	Meaning
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)

3.2.2.2.6 SyncOperation

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

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

Figure 3.21 SyncOperation Pseudocode Function

```
SyncOperation(stype)

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

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

endfunction SyncOperation
```

3.2.2.3 Floating Point Functions

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

3.2.2.3.1 ValueFPR

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

Figure 3.22 ValueFPR Pseudocode Function

```
value ← ValueFPR(fpr, fmt)

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

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


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
    case fmt of
        S, W, UNINTERPRETED_WORD:
            valueFPR ← FPR[fpr]

        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:
            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.

3.2.2.3.2 StoreFPR

Figure 3.23 StoreFPR Pseudocode Function

```
StoreFPR (fpr, fmt, value)

    /* fpr:   The FPR number */
    /* fmt:   The format of the data, one of: */
    /*        S, D, W, L, PS, */
    /*        OB, QH, */
    /*        UNINTERPRETED_WORD, */
    /*        UNINTERPRETED_DOUBLEWORD */
    /* value: The formattted value to be stored into the FPR */

    /* The UNINTERPRETED values are used to indicate that the datatype */
    /* is not known as, for example, in LWC1 and LDC1 */


    case fmt of
        S, W, UNINTERPRETED_WORD:
            FPR[fpr] ← value

        D, UNINTERPRETED_DOUBLEWORD:
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
            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:
            if (FP32RegistersMode = 0) then
                UNPREDICTABLE
            else
                FPR[fpr] ← value
            endif

    endcase

endfunction StoreFPR
```

3.2.2.3.3 CheckFPException

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

Figure 3.24 CheckFPException Pseudocode Function

```
CheckFPException()

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

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

endfunction CheckFPException
```

3.2.2.3.4 FPConditionCode

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

Figure 3.25 FPConditionCode Pseudocode Function

```
tf ←FPConditionCode(cc)

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

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

    if cc = 0 then
        FPConditionCode ← FCSR23
    else
        FPConditionCode ← FCSR24+cc
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
    endif

endfunction FPConditionCode
```

3.2.2.3.5 SetFPConditionCode

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

Figure 3.26 SetFPConditionCode Pseudocode Function

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

endfunction SetFPConditionCode
```

3.2.2.4 Pseudocode Functions Related to Sign and Zero Extension

3.2.2.4.1 Sign extension and zero extension in pseudocode

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

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

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

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

and

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

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

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

Figure 3.27 sign_extend Pseudocode Functions

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

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

The earlier examples can be expressed as

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

and

```
sign_extend(temp31..0)
= sign_extend.32(temp)
```

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

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

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

3.2.2.4.2 memory_address

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

Figure 3.28 memory_address Pseudocode Function

```
function memory_address(ea)
   return ea
end function
```

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

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

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

Figure 3.29 Instruction Fetch Implicit memory_address Wrapping

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

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

Figure 3.30 AddressTranslation implicit memory_address Wrapping

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

In addition to its use in instruction pseudocode,

3.2.2.5 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

3.2.2.5.1 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 3.31 SignalException Pseudocode Function

```
SignalException(Exception, argument)

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

endfunction SignalException
```

3.2.2.5.2 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 3.32 SignalDebugBreakpointException Pseudocode Function

```
SignalDebugBreakpointException()

endfunction SignalDebugBreakpointException
```

3.2.2.5.3 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 3.33 SignalDebugModeBreakpointException Pseudocode Function

```
SignalDebugModeBreakpointException()

endfunction SignalDebugModeBreakpointException
```

3.2.2.5.4 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 3.34 NullifyCurrentInstruction PseudoCode Function

```
NullifyCurrentInstruction()
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
endfunction NullifyCurrentInstruction
```

3.2.2.5.5 PolyMult

The PolyMult function multiplies two binary polynomial coefficients.

Figure 3.35 PolyMult Pseudocode Function

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

   PolyMult ← temp

endfunction PolyMult
```

3.3 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.

3.4 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 44 for a description of the *op* and *function* subfields.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Chapter 4

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

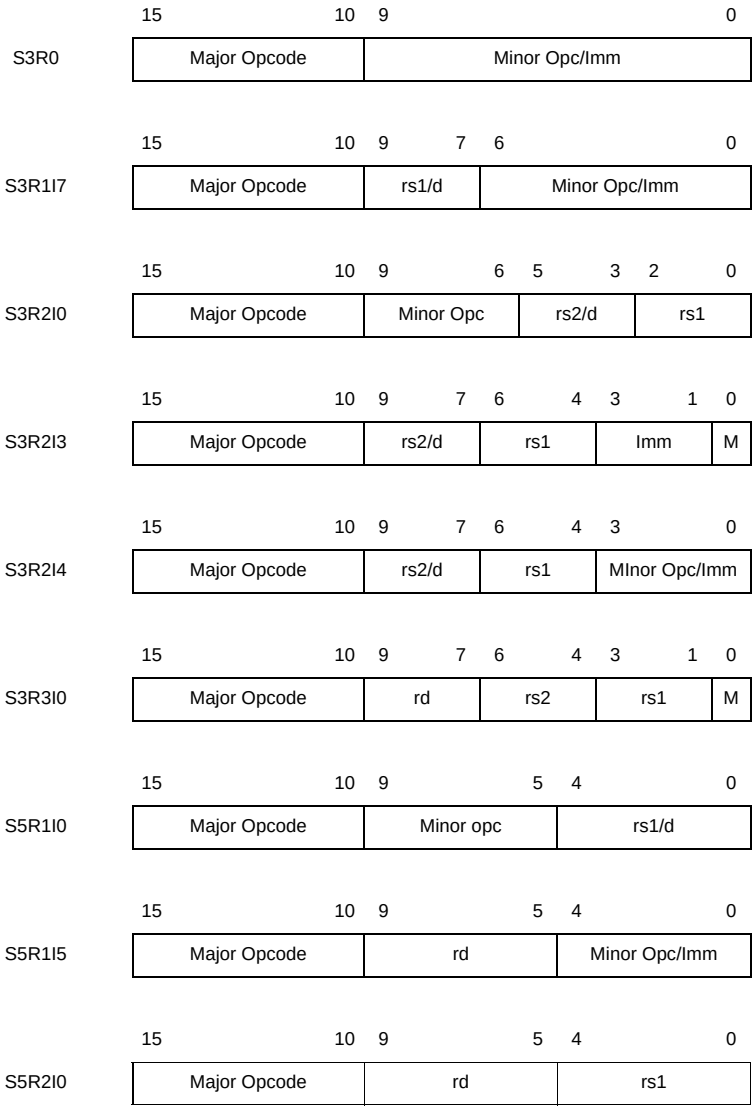
The name ‘immediate field’ as used here includes the address offset field for branches and load/store instructions as well as the jump target field.

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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Figure 4.1 16-Bit Instruction Formats



Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Figure 4.2 32-Bit Instruction Formats

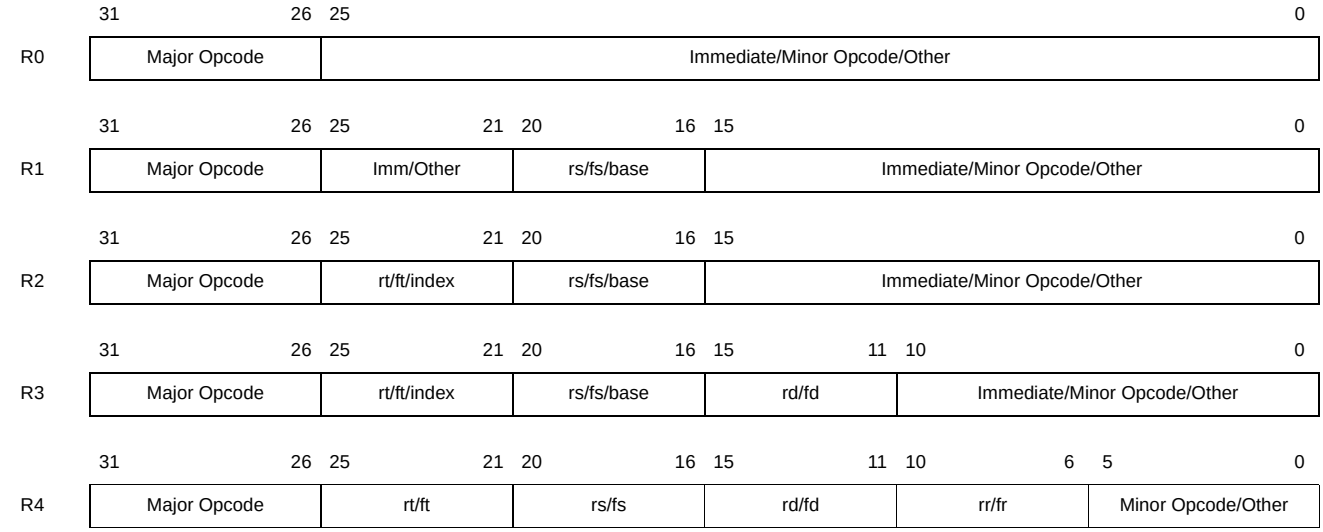
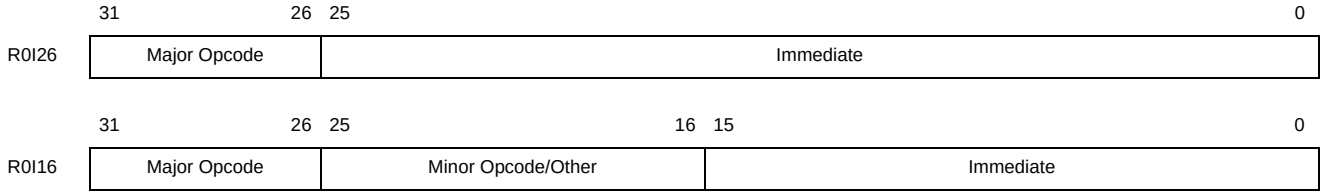
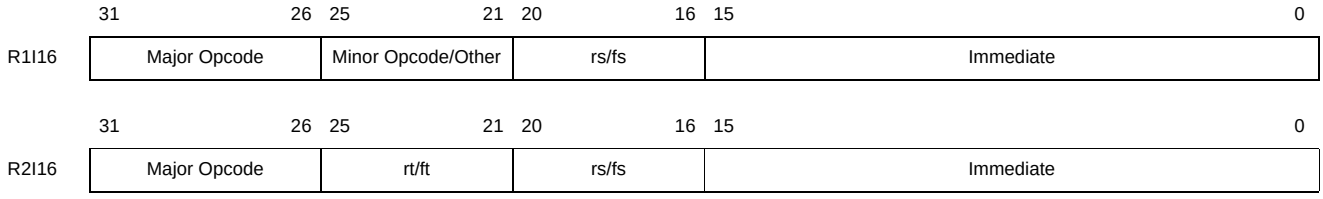


Figure 4.3 Immediate Fields within 32-Bit Instructions

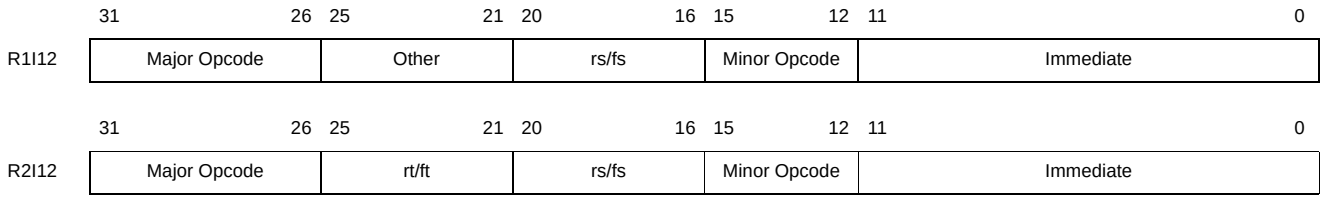
32-bit instruction formats with 26-bit immediate fields:



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



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



Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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.

- 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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

- 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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Chapter 5

microMIPS Instruction Set

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

In the 16-bit category:

- Frequent MIPS 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:

- 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

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 347, 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 MIPS Instructions

These are frequent MIPS 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 MIPS instruction.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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 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.

The instructions in the following tables are pre-Release 6 instructions. Refer to Section 2.7 "Release 6 of the MIPS Architecture" to understand which instructions have been removed in Release 6.

Table 5.1 16-Bit Re-encoding of Frequent MIPS 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
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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 5.1 16-Bit Re-encoding of Frequent MIPS 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
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
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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

5.1.2 Frequent MIPS Instruction Sequences

These 16-bit instructions are equivalent to frequently-used short sequences of MIPS instructions. The instruction-specific register and immediate value selection are shown in Table 5.3.

Table 5.2 16-Bit Re-encoding of Frequent MIPS 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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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 *ArchDefs.h*)	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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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 *ArchDefs.h*)	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 *ArchDefs.h*)	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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 5.6 16-Bit Instruction Implicit General-Purpose Registers

16-Bit Register Encoding	32-Bit MIPS Register Encoding	Symbolic Name (From *ArchDefs.h*)	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

The instructions in the following tables are pre-Release 6 instructions. Refer to Section 2.7 "Release 6 of the MIPS Architecture" to understand which instructions have been removed in Release 6.

5.3.1 New 32-bit instructions

The following table lists the 32-bit instructions introduced in the microMIPS ISA. Only instructions introduced prior to Release 6 are included in this table. JALRS, JALRS.HB, JALS, and JALX have been removed in Release 6.

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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
JALRS.HB	POOL32A	2:5 bit	0	5			16	Jump and Link Register with Hazard Barrier, Short Delay Slot
JALS	JALS32	0	26				0	Jump and Link, Short Delay Slot
JALX	JALX		26	5		0	5	Jump and Link Exchange
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
SWP	POOL32B	2:5 bit	12			0	4	Load Word Pair
SWM32	POOL32B	1:5bits	12		5	0	4	Store Word Multiple

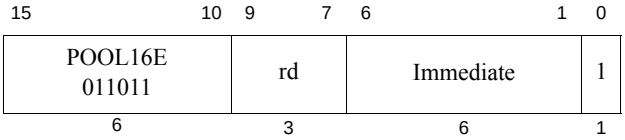
Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

5.4 microMIPS Instructions

This section describes instructions unique to microMIPS.

Only instructions supported in Release 6 are provided. Section 2.7, "Release 6 of the MIPS Architecture," lists instructions that have been added, removed and recoded in Release 6.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[29] + zero_extend(immediate << 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 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:

```
temp ← GPR[29] + zero_extend(immediate || 0^2)
GPR[rd] ← temp
```

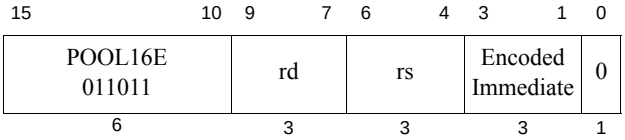
Exceptions:

None

Programming Notes:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[rs] + sign_extend(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 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.

Operation:

```
temp ← GPR[rs] + sign_extend(decoded immediate)
GPR[rd] ← temp
```

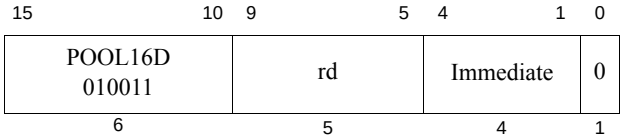
Exceptions:

None

Programming Notes:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[rd] + sign_extend(immediate)`

The 4-bit *immediate* field is sign-extended and then added to the 32-bit value in GPR *rd. T*he 32-bit arithmetic result is 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	0xfff8
9	0x9	-7	0xfff9
10	0xa	-6	0xfffa
11	0xb	-5	0xfffb
12	0xc	-4	0xfffc
13	0xd	-3	0xfffd
14	0xe	-2	0xfffe
15	0xf	-1	0xffff

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Restrictions:

Operation:

```
temp ← GPR[rd] + sign_extend(immediate)
GPR[rd] ← temp
```

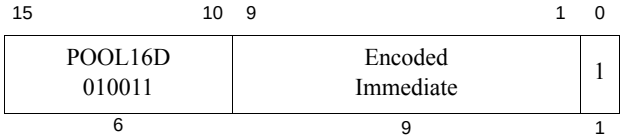
Exceptions:

None

Programming Notes:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[29] + sign_extend(decoded immediate << 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 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:

Operation:

```
temp ← GPR[29] + sign_extend(decoded immediate || 0^2)
GPR[29] ← temp
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

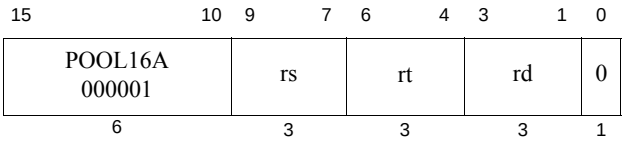
Exceptions:

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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `ADDU16 rd, rs, rt` **microMIPS**

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

To add 32-bit integers

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

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

Availability:

This instruction has been recoded for Release 6.

Operation:

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

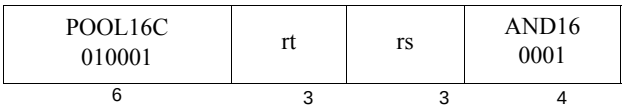
Exceptions:

None

Programming Notes:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `AND16 rt, rs` **microMIPS**

Purpose: And (16-bit instr size)

To do a bitwise logical AND

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

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

Restrictions:

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

Availability:

This instruction has been recoded for Release 6.

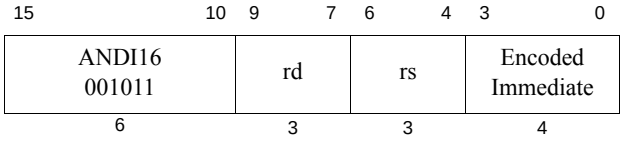
Operation:

```
GPR[rt] ← GPR[rs] and GPR[rt]
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[rs] AND zero_extend(decoded immediate)`

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.

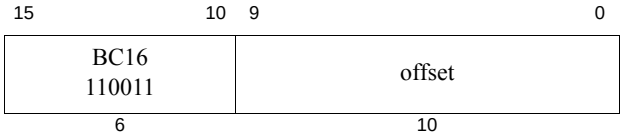
Operation:

```
GPR[rd] ← GPR[rs] and zero_extend(decoded immediate)
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `BC16 offset` **microMIPS**

Purpose: Unconditional Branch Compact (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) to form a PC-relative effective target address.

Compact branches do not have delay slots. The instruction after the branch is NOT executed when the branch is taken.

Restrictions:

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

Operation:

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

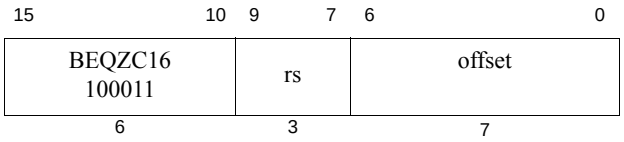
Exceptions:

None

Programming Notes:

With the 11-bit signed instruction offset, the branch range is ± 1 Kbytes. Use jump (JRC16 or JIC) or 32-bit branch instructions to branch to addresses outside this range.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `BEQZC16 rs, offset` **microMIPS**

Purpose: Branch on Equal to Zero Compact (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) to form a PC-relative effective target address.

If the contents of GPR *rs* equals zero, branch to the effective target address.

Compact branches do not have delay slots. The instruction after the branch is NOT executed if the branch is taken.

Restrictions:

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

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

Operation:

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

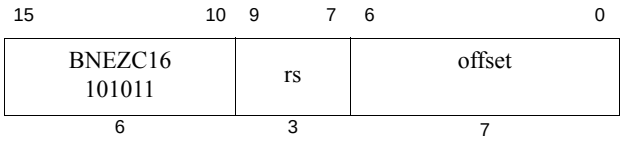
Exceptions:

None

Programming Notes:

With the 8-bit signed instruction offset, the conditional branch range is ± 64 Bytes. Use 32-bit branch, jump (JRC16 or JIC) instructions to branch to addresses outside this range.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `BNEZC16 rs, offset` **microMIPS**

Purpose: Branch on Not Equal to Zero Compact (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), to form a PC-relative effective target address.

If the contents of GPR *rs* does not equal zero, branch to the effective target address.

Compact branches do not have delay slots. The instruction after the branch is NOT executed if the branch is taken.

Restrictions:

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

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

Operation:

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

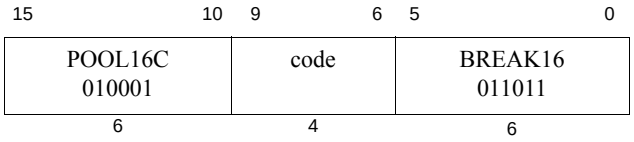
Exceptions:

None

Programming Notes:

With the 8-bit signed instruction offset, the conditional branch range is ± 64 Bytes. Use 32-bit branch, jump (JRC16 or JIC) instructions to branch to addresses outside this range.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Availability:

This instruction has been recoded for Release 6.

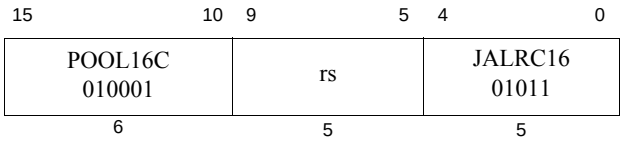
Operation:

```
SignalException(Breakpoint)
```

Exceptions:

Breakpoint

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `JALRC16 rs` **microMIPS Release 6**

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

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

Description: `GPR[31] ← return_addr, PC ← GPR[rs]`

For processors that do not implement the MIPS ISA:

- Jump to the effective target address in GPR *rs*. Bit 0 of GPR *rs* is interpreted as the target ISA Mode: if this bit is 0, signal an Address Error exception when the target instruction is fetched because this target ISA Mode is not supported. Otherwise, set bit 0 of the target address to zero, and fetch the instruction.

For processors that do implement the MIPS ISA:

- Jump to the effective target address in GPR *rs*. Set the ISA Mode bit to the value in GPR *rs* bit 0. Set bit 0 of the target address to zero. If the target ISA Mode bit is 0 and the target address is not 4-byte aligned, an Address Error exception will occur when the target instruction is fetched.

Place the return address link in GPR *r31*. The return link is the address of the first instruction following the branch, where execution continues after a procedure call.

Compact jumps do not have delay slots. The instruction after the jump is NOT executed when the jump is executed.

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 MIPS and if the ISAMode bit of the target is MIPS (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 MIPS ISA, if the intended target ISAMode is MIPS (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

Operation:

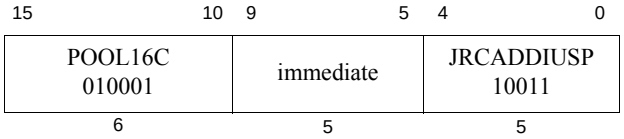
```
temp ← GPR[rs]
   GPR[31] ← PC + 4
if Config3ISA = 1 then
       PC ← temp
   else
       PC ← tempGPRLEN-1..1 || 0
       ISAMode ← temp0
   endif
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `JRCADDIUSP decoded_immediate` **microMIPS Release 6**

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

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

Description: `PC ← GPR[ra]; SP ← SP + zero_extend(Immediate << 2)`

For processors that do not implement the MIPS ISA:

- Jump to the effective target address in GPR *rs*. Bit 0 of GPR *rs* is interpreted as the target ISA Mode: if this bit is 0, signal an Address Error exception when the target instruction is fetched because this target ISA Mode is not supported. Otherwise, set bit 0 of the target address to zero, and fetch the instruction.

For processors that do implement the MIPS ISA:

- Jump to the effective target address in GPR *rs*. Set the ISA Mode bit to the value in GPR *rs* bit 0. Set bit 0 of the target address to zero. If the target ISA Mode bit is 0 and the target address is not 4-byte aligned, an Address Error exception will occur when the target instruction is fetched.

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 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.

Compact jumps do not have delay slots. The instruction after the jump is NOT executed when the jump is executed.

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 MIPS and the ISAMode bit of the target address is MIPS (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 MIPS ISA, if the intended target ISAMode is MIPS (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

Operation:

```
PC ← GPR[31]GPRLEN-1..1 || 0
if ( Config3ISA > 1 )
   ISAMode ← GPR[31]0
endif
temp ← GPR[29] + zero_extend(immediate || 02)
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

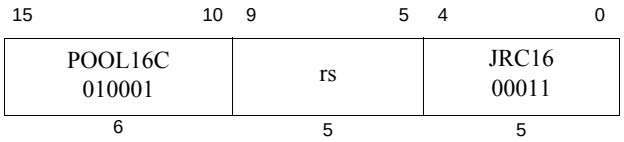
```
GPR[29] ← temp
```

Exceptions:

None.

Programming Notes:

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `JRC16 rs` **microMIPS Release 6**

Purpose: Jump Register Compact (16-bit instr size)

To execute a branch to an instruction address in a register

Description: `PC ← GPR[rs]`

For processors that do not implement the MIPS ISA:

- Jump to the effective target address in GPR *rs*. Bit 0 of GPR *rs* is interpreted as the target ISA Mode: if this bit is 0, signal an Address Error exception when the target instruction is fetched because this target ISA Mode is not supported. Otherwise, set bit 0 of the target address to zero, and fetch the instruction.

For processors that do implement the MIPS ISA:

- Jump to the effective target address in GPR *rs*. Set the ISA Mode bit to the value in GPR *rs* bit 0. Set bit 0 of the target address to zero. If the target ISA Mode bit is 0 and the target address is not 4-byte aligned, an Address Error exception will occur when the target instruction is fetched.

Compact jumps do not have delay slots. The instruction after the jump is NOT executed when the jump is executed.

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 MIPS and the ISAMode bit of the target address is MIPS (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 MIPS ISA, if the intended target ISAMode is MIPS (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

Compact jumps do not have delay slots. The instruction after the jump is NOT executed when the jump is executed.

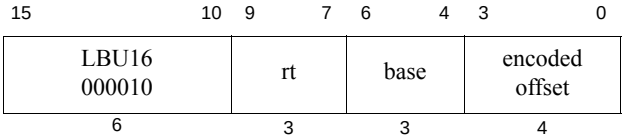
Operation:

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

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← 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 ← Decode(encoded_offset)
```

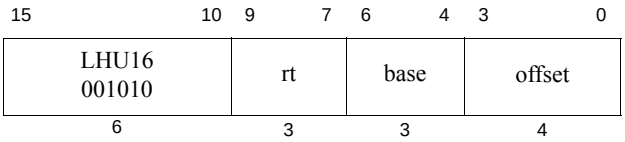

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
vAddr ← sign_extend(decoded_offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..  || (pAddr_..0 xor ReverseEndian)
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_..0 xor BigEndianCPU
GPR[rt] ← zero_extend(memword_7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `LHU16 rt, left_shifted_offset(base)` **microMIPS**

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

To load a halfword from memory as an unsigned value

Description: `GPR[rt] ← memory[GPR[base] + (offset × 2)]`

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 4-bit unsigned *offset* is left shifted by one bit and then 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 the least-significant bit of the address is non-zero, an Address Error exception occurs.

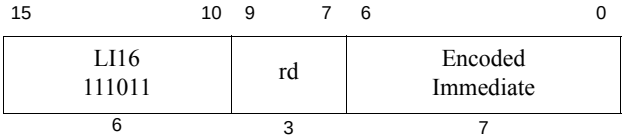
Operation:

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

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← 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 Immeidiate 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.

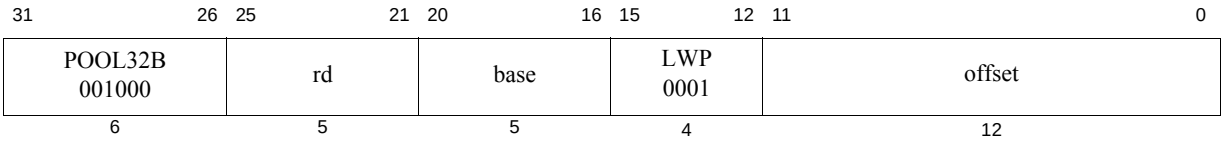
Operation:

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

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `LWP rd, offset(base)` **microMIPS**

Purpose: Load Word Pair

To load two consecutive words from memory

Description: `GPR[rd], GPR[rd+1] ← memory[GPR[base] + 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.

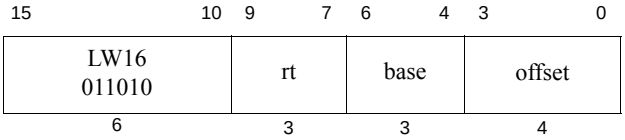
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rd] ← memword
vAddr ← sign_extend(offset) + GPR[base] + 4
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rd+1]← memword
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← memory[GPR[base] + (offset × 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.

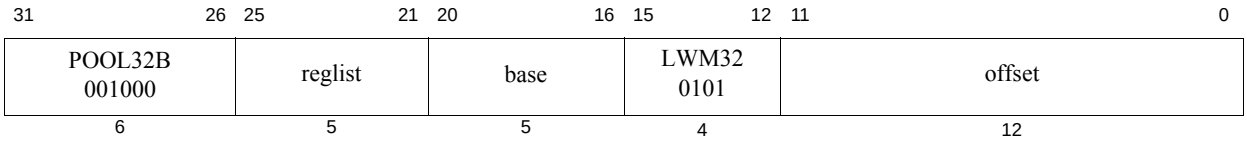
Operation:

```
vAddr ← sign_extend(offset|| 0^2) + GPR[base]
if vAddr_1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `LWM32 {sre16, } {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.

***reglist* 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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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.

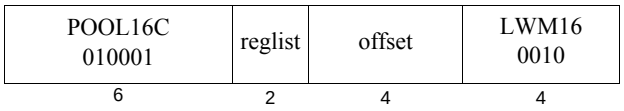
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
for i←0 to fn(reglist)
   (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
   memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
   GPR[gpr(reglist,i)] ← memword
   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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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.

***reglist* 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.

Operation:

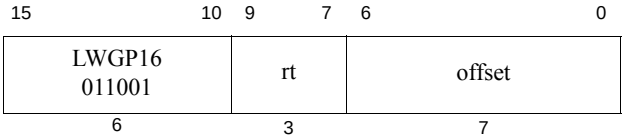
```
vAddr ← zero_extend(offset||0^2) + GPR[sp]
if vAddr_1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
for i← 0 to fn(reglist)
   (pAddr, CCA) ←AddressTranslation (vAddr, DATA, LOAD)
   memword ←LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
   GPR[gpr(reglist,i)] ←memword
   vAddr ←vAddr + 4
endfor

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


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← memory[GPR[28] + (offset × 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.

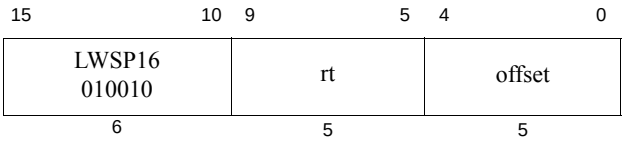
Operation:

```
vAddr ← sign_extend(offset|| 0^2) + GPR[28]
if vAddr_1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← memory[GPR[29] + (offset × 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.

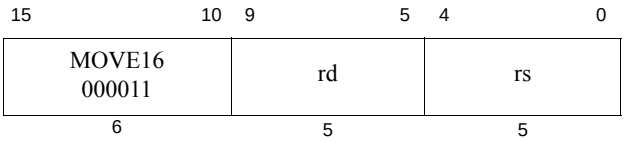
Operation:

```
vAddr ← zero_extend(offset|| 0^2) + GPR[29]
if vAddr_1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `MOVE16 rd, rs` **microMIPS**

Purpose: Move Register (16-bit instr size)

To copy one GPR to another GPR.

Description: `GPR[rd] ← GPR[rs]`

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

Restrictions:

None

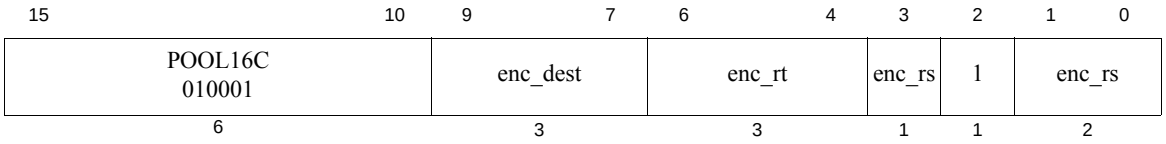
Operation:

```
GPR[rd] ← GPR[rs]
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Move a Pair of Registers

To copy two GPRs to another two GPRs.

Description: `GPR[rd] ← GPR[rs]; GPR[re] ← 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 *rd* (Decimal)	Decoded Value of *re* (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 rt (or rs) (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
7	0x7	20	s4

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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.

Operation:

```
GPR[rd] ← GPR[rs]; GPR[re] ← GPR[rt]
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Release 6

POOL16C 010001	rt	rs	NOT16 0000
6	3	3	4

Format: `NOT16 rt, rs` **microMIPS**

Purpose: Invert (16-bit instr size)

To do a bitwise logical inversion.

Description: `GPR[rt] ← GPR[rs] 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.

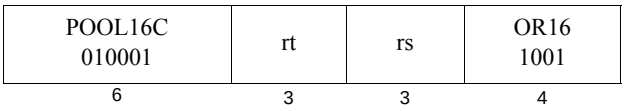
Operation:

```
GPR[rt] ← GPR[rs] xor 0xffffffff
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `OR16 rt, rs` **microMIPS32**

Purpose: Or (16-bit instr size)

To do a bitwise logical OR

Description: `GPR[rt] ← GPR[rs] or GPR[rt]`

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

Restrictions:

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

Availability:

This instruction has been recoded for Release 6.

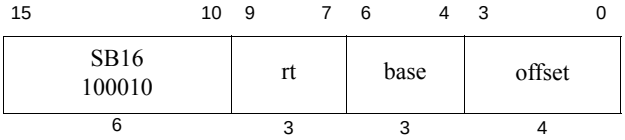
Operation:

```
GPR[rt] ← GPR[rs] or GPR[rt]
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SB16 rt, offset(base)` **microMIPS**

Purpose: Store Byte (16-bit instr size)

To store a byte to memory

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

The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 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.

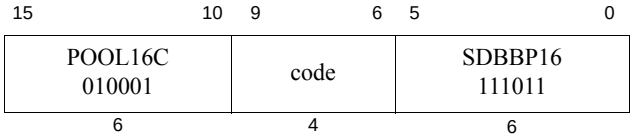
Operation:

```
vAddr ← zero_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1.. || (pAddr_..0 xor ReverseEndian)
bytesel ← vAddr_..0 xor BigEndianCPU
dataword ← GPR[rt]_-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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:

Availability:

This instruction has been recoded for Release 6.

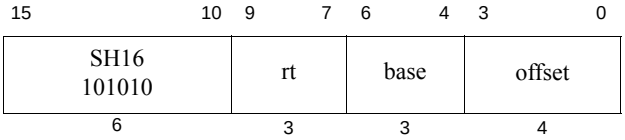
Operation:

```
If Debug_DM = 0 then
   SignalDebugBreakpointException()
else
   SignalDebugModeBreakpointException()
endif
```

Exceptions:

Debug Breakpoint Exception
Debug Mode Breakpoint Exception

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SH16 rt, left_shifted_offset(base)` **microMIPS**

Purpose: Store Halfword (16-bit instr size)

To store a halfword to memory

Description: `memory[GPR[base] + (offset ×2)] ← GPR[rt]`

The least-significant 16-bit halfword of register *rt* is stored in memory at the location specified by the aligned effective address. The 4-bit unsigned *offset* is left shifted by one bit and then 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.

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

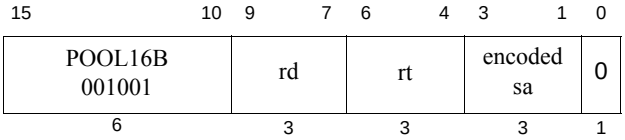
Operation:

```
vAddr ← zero_extend(offset|| 0) + GPR[base]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1.. || (pAddr..0 xor (ReverseEndian || 0))
bytesel ← vAddr..0 xor (BigEndianCPU || 0)
dataword ← GPR[rt]-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, HALFWORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[rt] << 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 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:

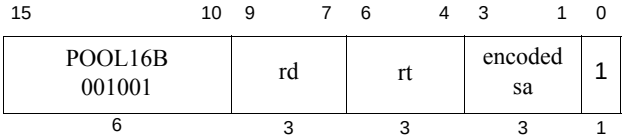
```
decoded_sa ← DECODE(encoded_sa)
s ← decoded_sa
temp ← GPR[rt](31-s)..0 || 0^s
GPR[rd] ← temp
```

Exceptions:

None

Programming Notes:

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[rt] >> 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 placed in GPR *rd*. The bit-shift amount is specified by . 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.

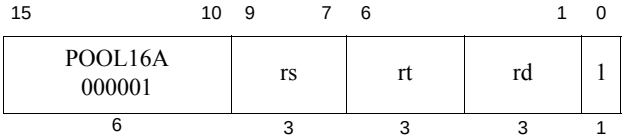
Operation:

```
decoded_sa ← DECODE(encoded_sa)
s ← decoded_sa
temp ← 0^s || GPR[rt]_31..s
GPR[rd] ← temp
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SUBU16 rd, rs, rt` **microMIPS**

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

To subtract 32-bit integers

Description: `GPR[rd] ← GPR[rs] – GPR[rt]`

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* and the 32-bit arithmetic result is 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.

Availability:

This instruction has been recoded for Release 6.

Operation:

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

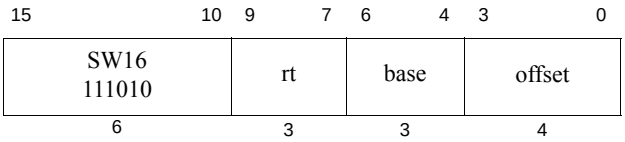
Exceptions:

None

Programming Notes:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SW16 rt, left_shifted_offset(base)` **microMIPS**

Purpose: Store Word (16-bit instr size)

To store a word to memory

Description: `memory[GPR[base] + (offset ×4)] ← GPR[rt]`

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 4-bit unsigned *offset* is left-shifted by two bits and then 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.

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.

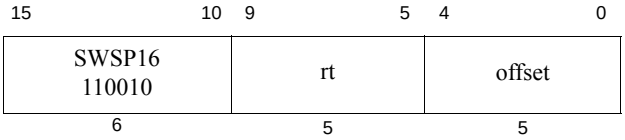
Operation:

```
vAddr ← zero_extend(offset || 0^2) + GPR[base]
if vAddr_1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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: `memory[GPR[29] + (offset ×4)] ← GPR[rt]`

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 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.

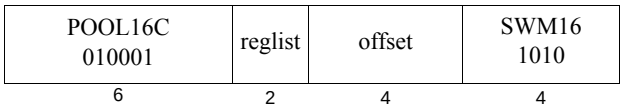
Operation:

```
vAddr ← zero_extend(offset|| 0^2) + GPR[29]
if vAddr_1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

***reglist* 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.

Availability:

This instruction has been recoded for Release 6.

Operation:

```
vAddr ← zero_extend(offset||0^2) + GPR[sp]
if vAddr_1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
for i← 0 to fn(reglist)
   (pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
   dataword ← GPR[gpr(reglist,i)]
   StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
   vAddr ← vAddr + 4
endfor

function fn(list)
```

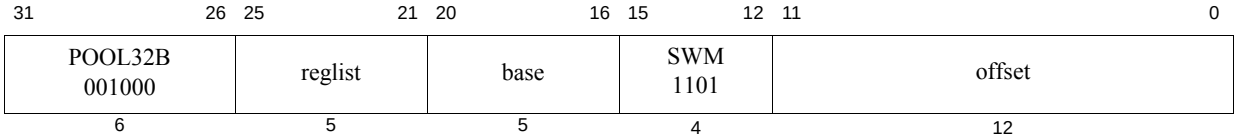

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
    fn ← number of entries in list - 1
endfunction
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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],...,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[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.

***reglist* 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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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.

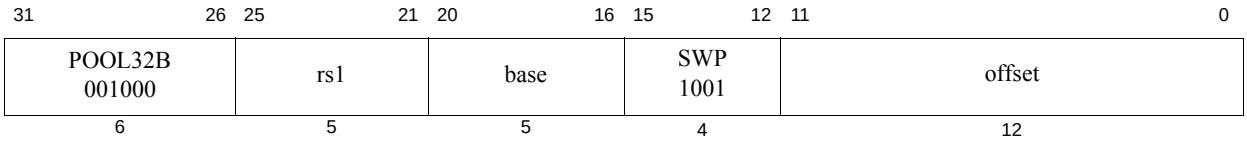
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
for i←0 to fn(reglist)
   (pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
   dataword ← GPR[gpr(reglist,i)]
   StoreMemory (CCA, WORD, dataword, 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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SWP rs1, offset(base)` **microMIPS**

Purpose: Store Word Pair

To store two consecutive words to memory

Description: `memory[GPR[base] + offset] ← GPR[rs1], GPR[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.

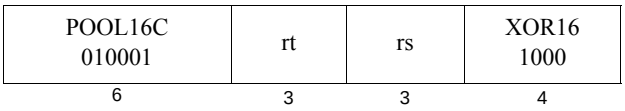
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rs1]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)

vAddr ← sign_extend(offset) + GPR[base] + 4
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rs1+1]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `XOR16 rt, rs` **microMIPS**

Purpose: Exclusive OR (16-bit instr size)

To do a bitwise logical Exclusive OR

Description: `GPR[rt] ← GPR[rs] XOR GPR[rt]`

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

Restrictions:

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

Availability:

This instruction has been recoded for Release 6.

Operation:

```
GPR[rt] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None

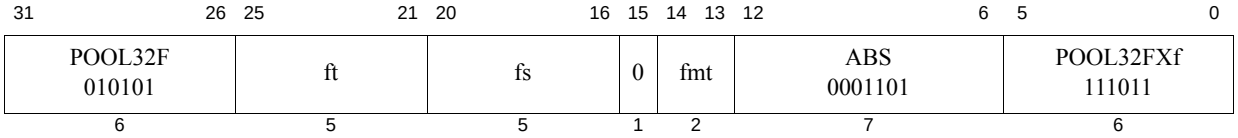
Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

5.5 MIPS Instructions

This section describes recoded 32-bit instructions from MIPS32 instruction sets specifically for use as part of the microMIPS instruction set.

Only instructions supported in Release 6 are provided. Section 2.7, "Release 6 of the MIPS Architecture," lists instructions that have been added, removed and recoded in Release 6.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `ABS.fmt`

`ABS.S ft, fs` **microMIPS**

`ABS.D ft, fs` **microMIPS**

Purpose: Floating Point Absolute Value

Description: `FPR[ft] ← abs(FPR[fs])`

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

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

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

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

Restrictions:

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

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

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

Availability and Compatibility:

Not applicable.

Operation:

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

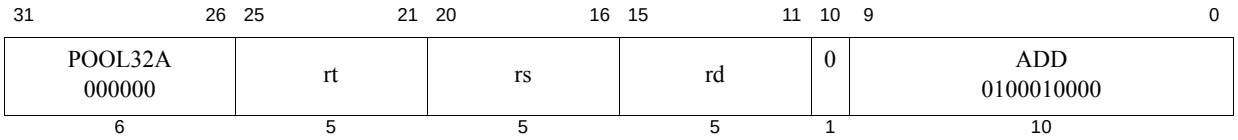
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Add Word

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

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

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

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

Restrictions:

Operation:

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

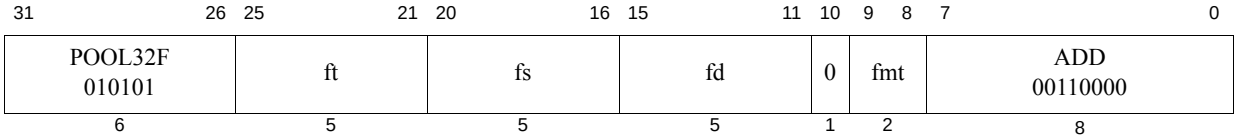
Exceptions:

Integer Overflow

Programming Notes:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `ADD.fmt`

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

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

Purpose: Floating Point Add

To add floating point values.

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

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

ADD.PS adds the upper and lower halves of FPR *fs* and FPR *ft* independently, and ORs together any generated exceptions.

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

Restrictions:

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

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

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

Availability and Compatibility:

Not applicable.

Operation:

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

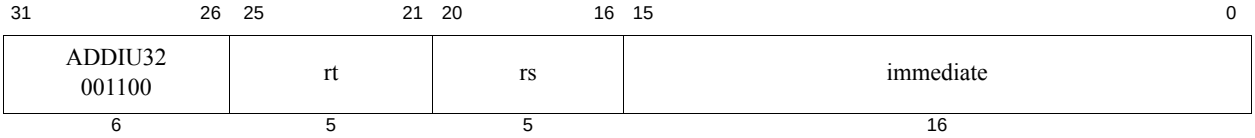
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Add Immediate Unsigned Word

To add a constant to a 32-bit integer.

Description: `GPR[rt] ← GPR[rs] + immediate`

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

No Integer Overflow exception occurs under any circumstances.

Restrictions:

Operation:

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

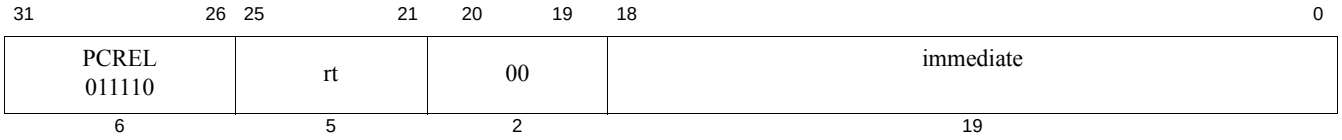
Exceptions:

None

Programming Notes:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `ADDIUPC rt,immediate` **microMIPS32 Release 6**

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

Description: `GPR[rt] ← ( PC & ~0x3 + sign_extend( immediate << 2 ) )`

This instruction performs a PC-relative address calculation. The 19-bit immediate is shifted left by 2 bits, sign-extended, and added to the address of the ADDIUPC instruction. The result is placed in GPR *rs*.

Restrictions:

None

Availability and Compatibility:

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

Operation:

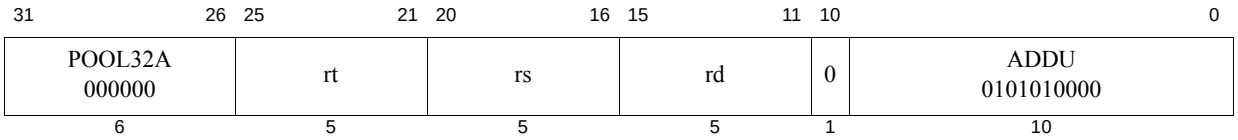
```
GPR[rd] ←  ( PC & ~0x3 + sign_extend( immediate << 2 ) )
```

Exceptions:

None

1. The term “unsigned” in this instruction mnemonic is a misnomer. “Unsigned:” here means “non-trapping”. It does not trap on a signed 32-bit overflow, the way pre-Release 6 ADD and ADDI do. ADDIUPC corresponds to unsigned ADDIU or ADDU, which do not trap on overflow, rather than ADDI, which traps on overflow.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Add Unsigned Word

To add 32-bit integers.

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

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

No Integer Overflow exception occurs under any circumstances.

Restrictions:

Operation:

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

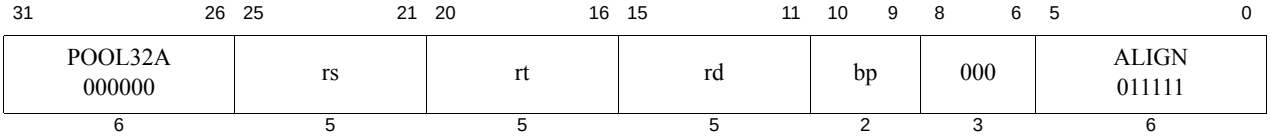
Exceptions:

None

Programming Notes:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `ALIGN`
`ALIGN rd,rs,rt,bp` **microMIPS32 Release 6**

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

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

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

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

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

Restrictions:

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

Availability and Compatibility:

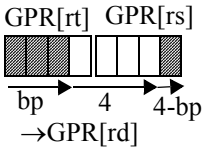
The ALIGN instruction is introduced by and required as of Release 6.

Programming Notes:

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

Graphically,

Figure 5.1 ALIGN operation (32-bit)



Operation:

```
tmp_rt_hi ← unsigned_word(GPR[rt]) << (8*bp)
tmp_rs_lo ← unsigned_word(GPR[rs]) >> (8*(4-bp))
tmp ← tmp_rt_hi or tmp_rt_lo
```

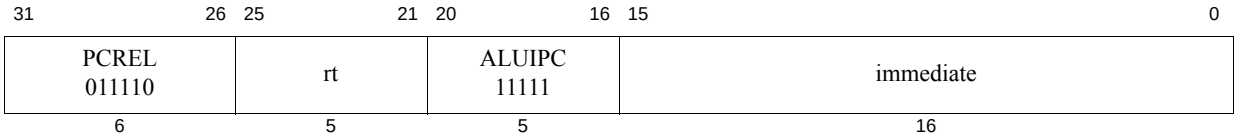

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
    GPR[rd] ← tmp
/* end of instruction */
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `ALUIPC rt,immediate` **microMIPS32 Release 6**

Purpose: Aligned Add Upper Immediate to PC

Description: `~0x0FFFF & ( PC + ( immediate << 16 ) )`

This instruction performs a PC-relative address calculation. The 16-bit immediate is shifted left by 16 bits, sign-extended, and added to the address of the ALUIPC instruction. The low 16 bits of the result are cleared, that is the result is aligned on a 64K boundary. The result is placed in GPR *rs*.

Restrictions:

None

Availability and Compatibility:

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

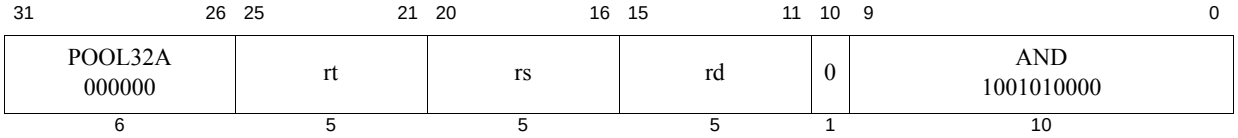
Operation:

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

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: And

To do a bitwise logical AND.

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

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

Restrictions:

None

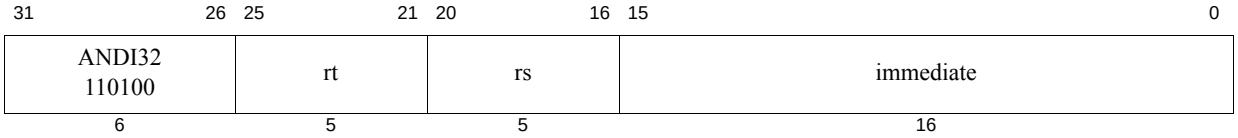
Operation:

```
GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: And Immediate

To do a bitwise logical AND with a constant

Description: `GPR[rt] ← GPR[rs] AND zero_extend(immediate)`

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

Restrictions:

None

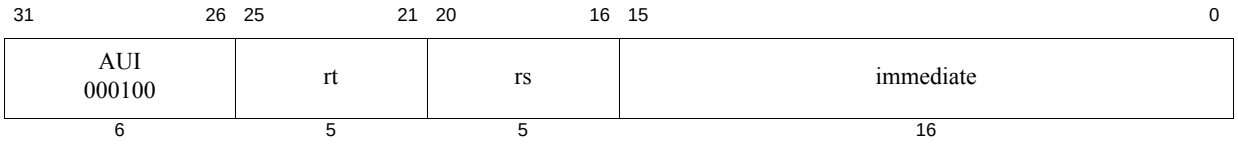
Operation:

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

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `AUI`

`AUI rt, rs immediate` **microMIPS32 Release 6**

Purpose: Add Immediate to Upper Bits

Add Upper Immediate

Description:

```
GPR[rt] ← GPR[rs] + (immediate << 16)
```

The 16 bit immediate is shifted left 16 bits, sign-extended, and added to the register `rs`, storing the result in `rt`.

Restrictions:

Availability and Compatibility:

AUI is introduced by and required as of Release 6.

Operation:

```
AUI:  GPR[rt] ← GPR[rs] + (immediate << 16)
```

Exceptions:

None.

In Release 6, LUI is an assembly idiom for AUI with rs=0.

Programming Notes:

AUI can be used to synthesize large constants in situations where it is not convenient to load a large constant from memory. To simplify hardware that may recognize sequences of instructions as generating large constants, AUI should be used in a stylized manner.

To create an integer:

```
LUI rd, imm_low
ORI rd, rd, imm_upper
```

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

```
AUI rtmp, rbase, imm_upper
LW rd, (rtmp)imm_low
```

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

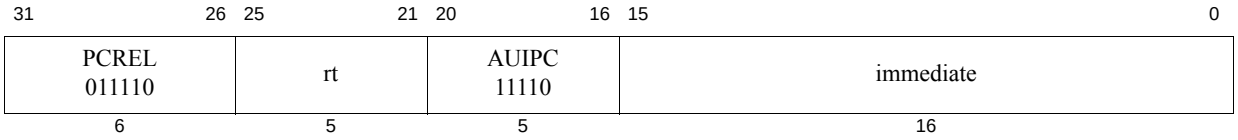
```
AUI   rtmp, rs,   imm_upper
ADDUI rd,   rtmp, imm_low
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `AUIPC rt, immediate` **microMIPS32 Release 6**

Purpose: Add Upper Immediate to PC

Description: `GPR[rt] ← ( PC & ~0x3 + ( immediate << 16 ) )`

This instruction performs a PC-relative address calculation. The 16-bit immediate is shifted left by 16 bits, sign-extended, and added to the address of the AUIPC instruction. The result is placed in GPR *rt*.

Restrictions:

None

Availability and Compatibility:

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

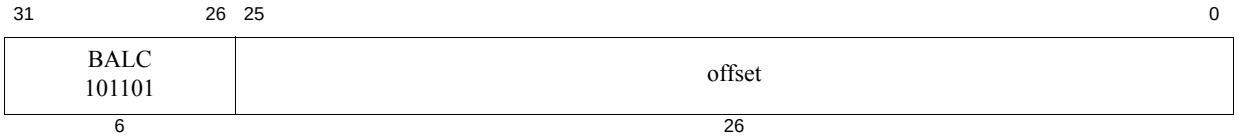
Operation:

```
GPR[rt] ← ( PC & ~0x3 + ( immediate << 16 ) )
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `BALC offset` **microMIPS32**

Purpose: Branch and Link, Compact

To do an unconditional PC-relative procedure call.

Description: `procedure_call (no delay slot)`

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

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

Compact branches do not have delay slots. The instruction after the branch is NOT executed when the branch is taken.

Restrictions:

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

Availability and Compatibility:

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

Exceptions:

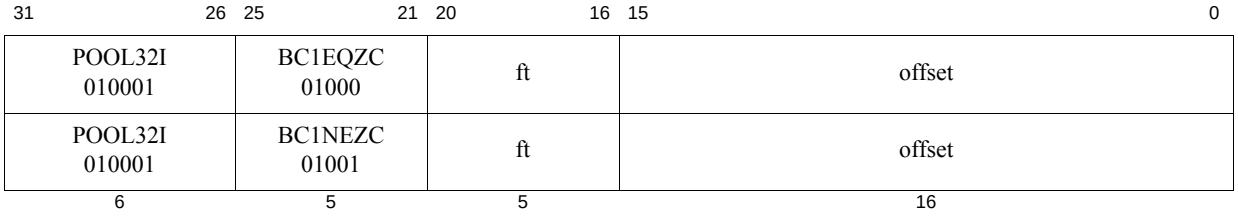
None[2]

Operation:

```
target_offset ← sign_extend( offset || 0[1] )
GPR[31] ← PC+4
PC ← PC+4 + sign_extend(target_offset)
```

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `BC1EQZC BC1NEZC`

`BC1EQZC ft, offset` **microMIPS32 Release 6**

`BC1NEZC ft, offset` **microMIPS32 Release 6**

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

BC1EQZC: Branch if Coprocessor 1 (FPU) Register Bit 0 is Equal to Zero

BC1NEZC: Branch if Coprocessor 1 (FPR) Register Bit 0 is Not Equal to Zero

Description:

```
BC1EQZC:  if FPR[ft].bit0 = 0 then branch
BC1NEZC:  if FPR[ft].bit0 ≠ 0 then branch
```

The condition is evaluated on FPU register *ft*.

- For BC1EQZC, the condition is true if and only if bit 0 of the FPU register *ft* is zero.
- For BC1NEZC, the condition is true if and only if bit 0 of the FPU register *ft* is non-zero.

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

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

Compact branches do not have delay slots. The instruction after the branch is NOT executed if the branch is taken.

Restrictions:

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

Because these instructions BC1EQZC and BC1NEZC do not depend on a particular floating point data type, they operate whenever Coprocessor 1 is enabled.

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

Availability and Compatibility:

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

Exceptions:

Coprocessor Unusable[2]

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Operation:

```
tmp ← ValueFPR(ft, UNINTERPRETED_WORD)
BC1EQZC: cond ← tmp.bit0 = 0
BC1NEZC: cond ← tmp.bit0 ≠ 0
if cond then
       target_PC ← ( PC+4 + sign_extend( offset << 1 )
       PC ←target_PC
```

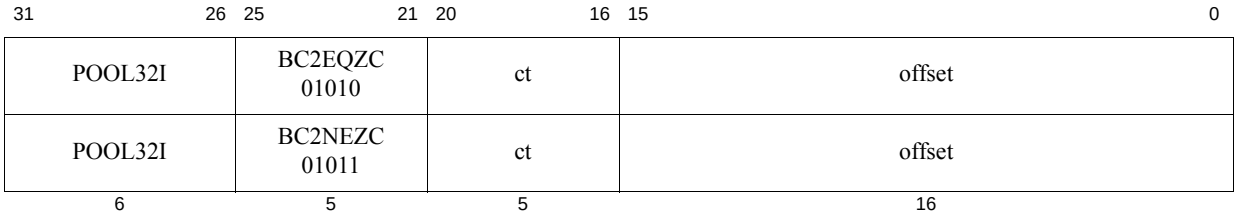
Programming Notes:

Release 6: These instructions, BC1EQZC and BC1NEZC, replace the pre-Release 6 instructions BC1F and BC1T. These Release 6 FPU branches depend on bit 0 of the scalar FPU register.

Note: BC1EQZC and BC1NEZC do not have a format or data type width. The same instructions are used for branches based on conditions involving any format, including 32-bit S (single precision) and W (word) format, and 64-bit D (double precision) and L (longword) format, as well as 128-bit MSA. The FPU scalar comparison instruction CMP.condn.fmt produces a mask of the same width as its format. For example, CMP.condn.S produces a 32-bit mask, with the upper 32-bits UNPREDICTABLE. Nevertheless, BC1EQZ and BC1NEQZ, considering only bit 0, operate as expected. That is, CMP.condn.fmt produces an allZeroes/allOnes truth mask of its format width, whereas the BC1* branches consume bit 0 only. Similarly for CLASS.fmt.

2. In Release 6, BC1EQZC and BC1NEZC are required, if the FPU is implemented. They must not signal a Reserved Instruction Exception. They can signal a Coprocessor Unusable Exception.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



31 – 26	25 – 21	20 – 16	15 – 0
POOL32I	BC2EQZC 01010	ct	offset
POOL32I	BC2NEZC 01011	ct	offset
6	5	5	16

Format: `BC2EQZC BC2NEZC`

`BC2EQZC ct, offset` **microMIPS32 Release 6**

`BC2NEZC ct, offset` **microMIPS32 Release 6**

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

BC2EQZC: Branch if Coprocessor 2 Condition (Register) is Equal to Zero

BC2NEZC: Branch if Coprocessor 2 Condition (Register) is Not Equal to Zero

Description:

```
BC2EQZC:  if COP2Condition[ct] = 0 then branch
BC2NEZC:  if COP2Condition[ct] ≠ 0 then branch
```

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

- For BC2EQZC if the coprocessor 2 condition is true the branch is taken.
- For BC2NEZC if the coprocessor 2 condition is false the branch is taken.

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

Compact branches do not have delay slots. The instruction after the branch is NOT executed if the branch is taken.

Restrictions:

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

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

Availability and Compatibility:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Operation:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
tmpcond ← Coprocessor2Condition(ct)
if BC2NEZC then
 tmpcond ← not(tmpcond)
endif

if tmpcond then
   PC ← PC+4 + sign_extend( immediate << 1 ) )
endif
```

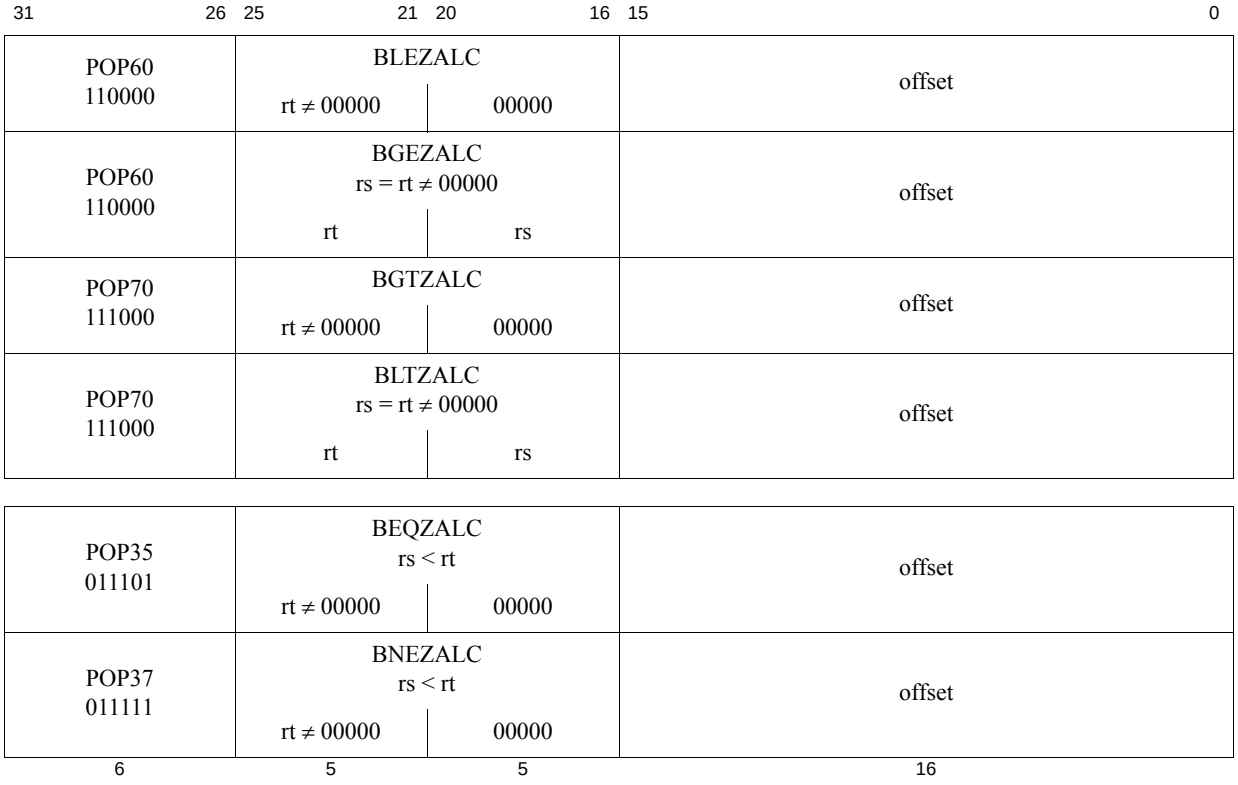
Implementation Notes:

As of Release 6 these instructions, BC2EQZC and BC2EQZC, replace the pre-Release 6 instructions BC2F and BC2T, which had a 3-bit condition code field (as well as nullify and true/false bits).

Release 6 makes all 5 bits of the `ct` condition code available to the coprocessor designer as a condition specifier.

A customer defined coprocessor instruction set can implement any sort of condition it wants. For example, it could implement up to 32 single-bit flags, specified by the 5-bit field `ct`. It could also implement conditions encoded as values in a coprocessor register (such as testing the least significant bit of a coprocessor register) as done by Release 6 coprocessor 1, the FPU, instructions BC1EQZ/BC1/NEZ.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

`BLEZALC rt, offset`	**microMIPS32 Release 6**
`BGEZALC rt, offset`	**microMIPS32 Release 6**
`BGTZALC rt, offset`	**microMIPS32 Release 6**
`BLTZALC rt, offset`	**microMIPS32 Release 6**
`BEQZALC rt, offset`	**microMIPS32 Release 6**
`BNEZALC rt, offset`	**microMIPS32 Release 6**

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

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

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

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

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

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

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

Description: `if condition(GPR[rt]) then procedure_call branch`

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

Places the return address link in GPR 31. The return link is the address of the instruction immediately following the branch, where execution continues after a procedure call. (Compact branch-and-links such as these have no delay

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

slots, see below.)

The return address link is unconditionally updated.

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

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

Compact branches do not have delay slots. The instruction after the branch is NOT executed when the branch is taken.

Restrictions:

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

Availability and Compatibility:

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

Exceptions:

None[2]

Operation:

```
GPR[31] ← PC+4
target_offset ← sign_extend( offset || 0[1] )

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

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

Programming Notes:

Software that performs incomplete instruction decode may incorrectly decode these new instructions, because of their

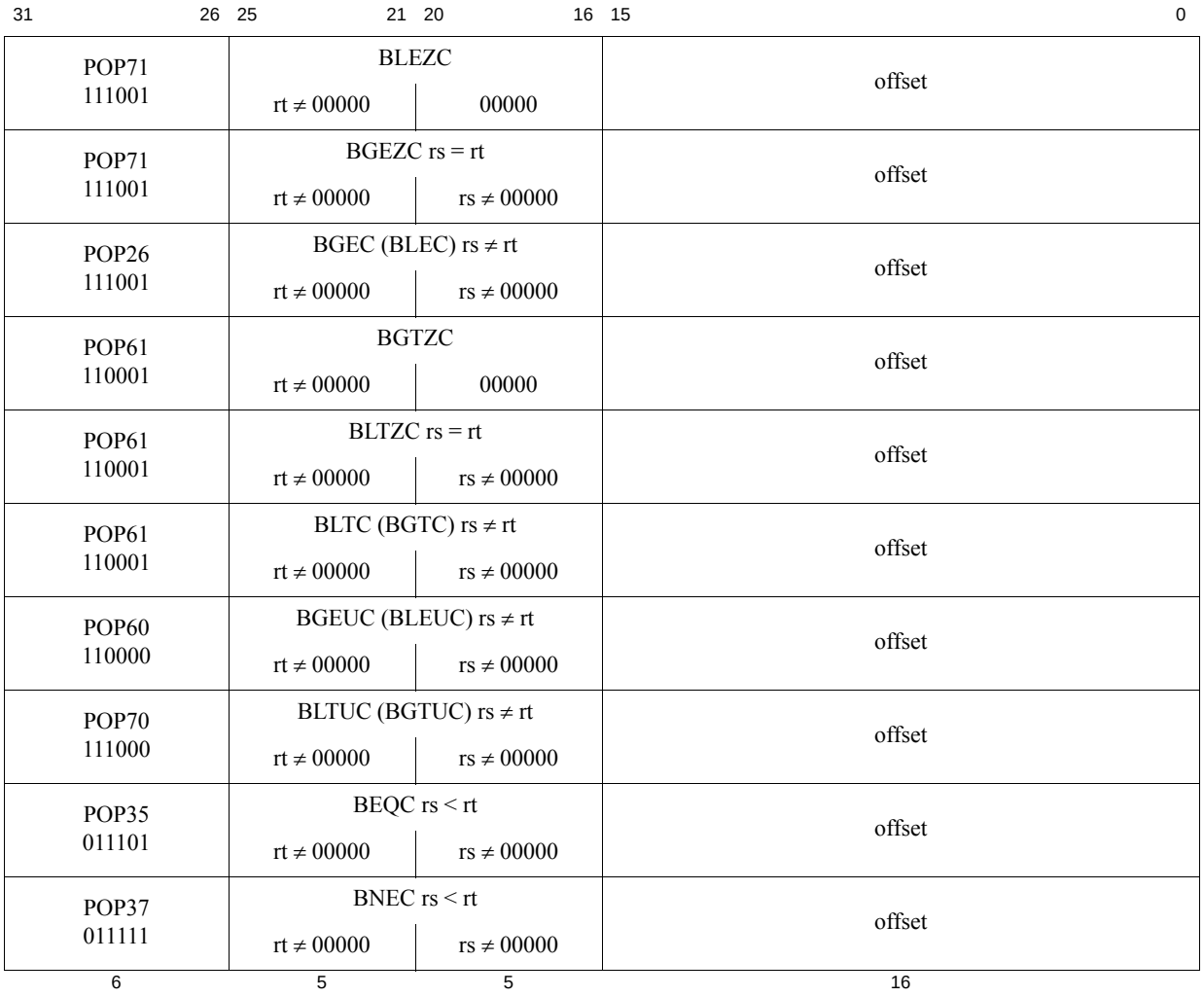
1. In this instruction description, PC refers to the PC of the current instruction. In some older instruction descriptions, PC is used inconsistently, often in association with the I: / I+1: notation related to delayed branches. For example, in Pre-Release 6 branch (B) PC refers to the address of the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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

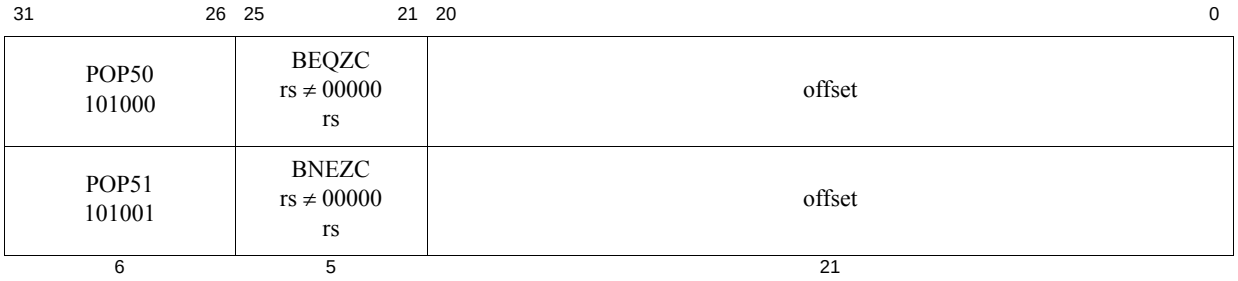
very tight encoding. For example, a disassembler might look only at the primary opcode field, instruction bits 31-26, to decode BLEZL without checking that the “rt” field is zero. Such software violated the pre-Release 6 architecture specification.

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



31–26	25–21	20–16	15–0
POP71 111001	BLEZC rt ≠ 00000	00000	offset
POP71 111001	BGEZC rs = rt rt ≠ 00000	rs ≠ 00000	offset
POP26 111001	BGEC (BLEC) rs ≠ rt rt ≠ 00000	rs ≠ 00000	offset
POP61 110001	BGTZC rt ≠ 00000	00000	offset
POP61 110001	BLTZC rs = rt rt ≠ 00000	rs ≠ 00000	offset
POP61 110001	BLTC (BGTC) rs ≠ rt rt ≠ 00000	rs ≠ 00000	offset
POP60 110000	BGEUC (BLEUC) rs ≠ rt rt ≠ 00000	rs ≠ 00000	offset
POP70 111000	BLTUC (BGTUC) rs ≠ rt rt ≠ 00000	rs ≠ 00000	offset
POP35 011101	BEQC rs < rt rt ≠ 00000	rs ≠ 00000	offset
POP37 011111	BNEC rs < rt rt ≠ 00000	rs ≠ 00000	offset
6	5	5	16



31–26	25–21	20–0
POP50 101000	BEQZC rs ≠ 00000 rs	offset
POP51 101001	BNEZC rs ≠ 00000 rs	offset
6	5	21

Format: `B<cond>C rs,rt, offset` **microMIPS32 Release 6**

Purpose: Compact compare-and-branch instructions

Format Details:

Equal/Not-Equal register-register compare and branch with 16-bit offset:

`BEQC rs,rt, offset` **microMIPS32 Release 6**

`BNEC rs,rt, offset` **microMIPS32 Release 6**

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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

```
BLTC rs,rt, offset                                  microMIPS32 Release 6
BGEC rs,rt, offset                                  microMIPS32 Release 6
```

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

```
BLTUC rs,rt, offset                                 microMIPS32 Release 6
BGEUC rs,rt, offset                                 microMIPS32 Release 6
```

Assembly idioms with reversed operands for signed/unsigned compare-and-branch:

```
BGTC  rt,rs, offset                                 Assembly Idiom
BLEC  rt,rs, offset                                 Assembly Idiom
BGTUC rt,rs, offset                                 Assembly Idiom
BLEUC rt,rs, offset                                 Assembly Idiom
```

Signed Compare register to Zero and branch with 16-bit offset:

```
BLTZC rt, offset          microMIPS32 Release 6BLEZC rt, rs, offsetmicroMIPS32 Release 6
BGEZC rt, offset                                    microMIPS32 Release 6
BGTZC rt, rs, offset                                microMIPS32 Release 6
```

Equal/Not-equal Compare register to Zero and branch with 21-bit offset:

```
BEQZC rt, rs, offset                                microMIPS32 Release 6
BNEZC rt, rs, offset                                microMIPS32 Release 6
```

Description: `if condition(GPR[rs] and/or GPR[rt]) then compact branch`

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

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

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

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

The conditions are as follows:

Equal/Not-equal register-register compare-and-branch with 16-bit offset:

BEQC: Compact branch if GPRs are equal
BNEC: Compact branch if GPRs are not equal

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

BLTC: Compact branch if GPR `rs` is less than GPR `rt`
BGEC: Compact branch if GPR `rs` is greater than or equal to GPR `rt`

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

BLTUC: Compact branch if GPR `rs` is less than GPR `rt`, unsigned
BGEUC: Compact branch if GPR `rs` is greater than or equal to GPR `rt`, unsigned

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Assembly Idioms with Operands Reversed:

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

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

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

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

BEQZC: Compact branch if GPR `rs` is equal to zero
BNEZC: Compact branch if GPR `rs` is not equal to zero

Restrictions:

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

Availability and Compatibility:

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

Exceptions:

None[2]

Operation:

```
target_offset ← sign_extend( offset || 0[1] )

/* Register-register compare and branch, 16 bit offset: */
/* Equal / Not-Equal */
BEQC: cond ← GPR[rs] = GPR[rt]
BNEC: cond ← GPR[rs] ≠ GPR[rt]
/* Signed */
BLTC: cond ← GPR[rs] < GPR[rt]
BGEC: cond ← GPR[rs] ≥ GPR[rt]
/* Unsigned: */
BLTUC: cond ← unsigned(GPR[rs]) < unsigned(GPR[rt])
BGEUC: cond ← unsigned(GPR[rs]) ≥ unsigned(GPR[rt])

/* Compare register to zero, small offset: */
BLTZC: cond ← GPR[rt] < 0
BLEZC: cond ← GPR[rt] ≤ 0
BGEZC: cond ← GPR[rt] ≥ 0
BGTZC: cond ← GPR[rt] > 0
/* Compare register to zero, large offset: */
BEQZC: cond ← GPR[rs] = 0
BNEZC: cond ← GPR[rs] ≠ 0
```

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
if cond then
  PC ← ( PC+4+ sign_extend( offset ) )
end if
```

Special Considerations:

See elsewhere for a complete overview of Release 6 instruction encodings. Brief notes related to compact branches such as these instructions:

In order to provide a full set of compare-and-branch instructions, Release 6 reduces some of the redundancy in instruction encoding related to the R0 register. This is indicated by constraints applied to the register encodings. For example, `BGEZC rs,rt` is encoded as `010110.rs.rt.offset16`, with constraints `rs≠00000`, `rt≠00000` and `rs=rt` (since `BGEZ` with both operands the same register is always true, hence redundant). Similarly, `BEQC rs,rt` has the constraints `rs≠00000`, `rt≠00000`, and `rs<rt` (since `BEQC` with both operands the same register is always true, hence redundant). Note that "`rs`" and "`rt`" in these constraints refer to register numbers as encoded within the instruction, not the values contained by the registers.

Table 5.17 below shows that the compact branches provide a full set of integer comparisons. It is necessary to reverse the operands of the signed and unsigned register-register compares in the shaded cases. Separate instructions are required for comparisons against 0, because the GPR[0] encoding is not allowed in the register-register comparisons, to save encoding space. Assembly idioms may provide the "reversed" instructions.

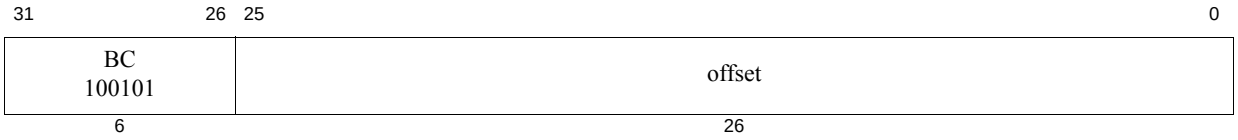
Table 5.17 Compact branches provide a full set of comparisons

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

Programming Notes:

Legacy software that performs incomplete instruction decode may incorrectly decode these new instructions, because of their very tight encoding. For example, a disassembler that looks only at the primary opcode field (instruction bits 31-26) to decode BLEZL without checking that the "rt" field is zero violates the pre-Release 6 architecture specification. Complete instruction decode allows reuse of pre-Release 6 BLEZL opcode for Release 6 conditional branches.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `BC offset` **microMIPS32 Release 6**

Purpose: Branch, Compact

Description: `PC ← PC+4 + sign_extend( offset << 1)`

The branch target is formed by sign extending the 26-bit offset field of the instruction shifted left by 1 bit (because microMIPS instructions may be 2 byte aligned), and adding it to the PC of the following instruction, such as adding it to the PC of the current instruction + the instruction length, PC+4.[1]

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

Restrictions:

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

Availability and Compatibility:

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

Exceptions:

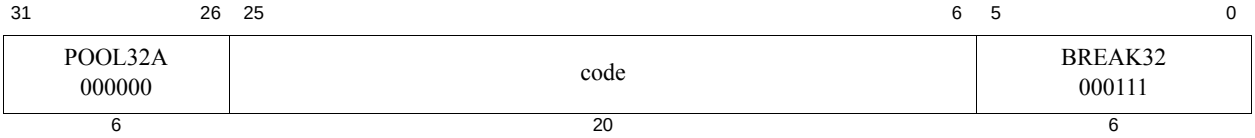
None[2]

Operation:

```
target_offset ← sign_extend( offset || 0[1] )
PC ← ( PC+4 + sign_extend(target_offset) )
```

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

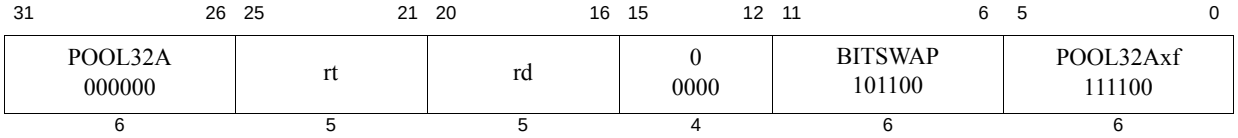
Operation:

```
SignalException(Breakpoint)
```

Exceptions:

Breakpoint

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `BITSWAP`
`BITSWAP rd,rt` **microMIPS32 Release 6**

Purpose: Swaps (reverses) bits in each byte

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

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

BITSWAP operates on all 4 bytes of a 32-bit GPR on a 32-bit CPU.

Restrictions:

BITSWAP: None.

Availability and Compatibility:

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

Operation:

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



       where

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

Exceptions:

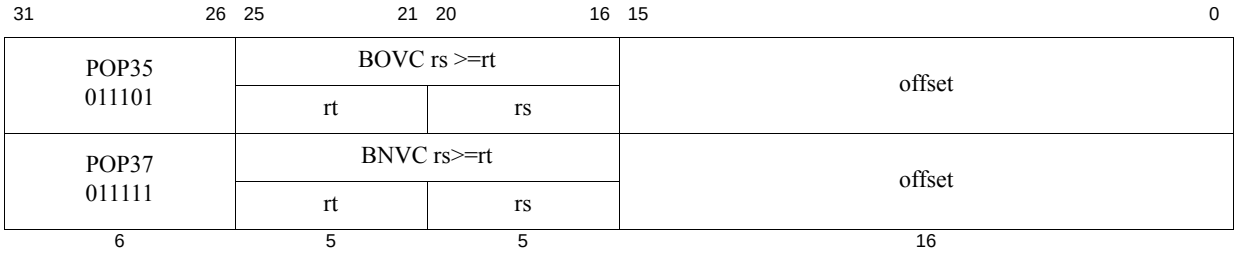
None

Programming Notes:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `BOVC BNVC`

`BOVC rt,rs, offset` **microMIPS32 Release 6**

`BNVC rt,rs, offset` **microMIPS32 Release 6**

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

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

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

Description: `branch if/if-not  NotWordValue(GPR[rs]+GPR[rt])`

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

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

The special case with `rt`=0 (for example, GPR[0]) is allowed.

The special case of rs=0 and rt=0 is allowed. BOVC never branches, while BNVC always branches.

Compact branches do not have delay slots. The instruction after the branch is NOT executed if the branch is taken.

Restrictions:

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

Availability and Compatibility:

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

Operation:

```
temp1 ← GPR[rs]
temp2 ← GPR[rt]
tempd ← temp1 + temp2 // wider than 32-bit precision
sum_overflow ← (tempd32 ≠ tempd31)
```

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

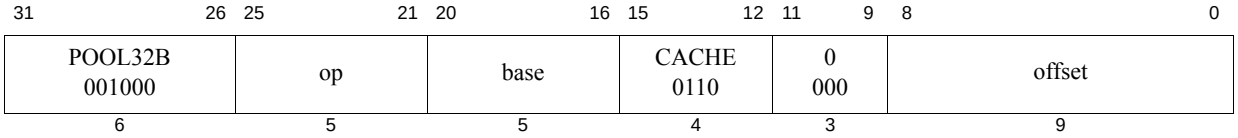
Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
BOVC: cond ← sum_overflow
BNVC: cond ← not( sum_overflow )

if cond then
   PC ← ( PC+4 + sign_extend( offset << 1 ) )
endif
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Perform Cache Operation

To perform the cache operation specified by op.

Description:

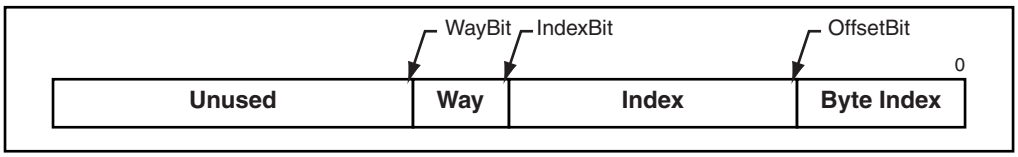
The 9-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

Table 5.18 Usage of Effective Address

Operation Requires an	Type of Cache	Usage of Effective Address
Address	Virtual	The effective address is used to address the cache. An address translation may or may not be performed on the effective address (with the possibility that a TLB Refill or TLB Invalid exception might occur)
Address	Physical	The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache

Index	N/A	The effective address is translated by the MMU to a physical address. It is implementation dependent whether the effective address or the translated physical address is used to index the cache. As such, an unmapped address (such as within kseg0) should always be used for cache operations that require an index. See the Programming Notes section below.
		
		Assuming that the total cache size in bytes is CS, the associativity is A, and the number of bytes per tag is BPT, the following calculations give the fields of the address which specify the way and the index:
		
		`OffsetBit ← Log2(BPT)`
		`IndexBit ← Log2(CS / A)`
		`WayBit ← IndexBit + Ceiling(Log2(A))`
		`Way ← Addr`$_{\texttt{WayBit-1..IndexBit}}$
		`Index ← Addr`$_{\texttt{IndexBit-1..OffsetBit}}$
		For a direct-mapped cache, the Way calculation is ignored and the Index value fully specifies the cache tag. This is shown symbolically in the figure below.

Figure 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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

operations (where the address is used to index the cache but need not match the cache tag), software must use unmapped addresses to avoid TLB exceptions. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS. This instruction never causes Execute-Inhibit nor Read-Inhibit exceptions.

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

As a result, a Cache Error exception may occur because 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. Also, 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 5.19 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

When implementing 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; must first operate on the smaller, inner-level cache. For example, a Hit_Writeback _Invalidate operation targeting the Secondary cache, must 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.

When implementing 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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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 5.20 Encoding of Bits [20:18] of the CACHE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b000	I	Index Invalidate	Index	Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.	Required
	D	Index Writeback Invalidate / Index Invalidate	Index	For a write-back cache: If the state of the cache block at the specified index is valid and dirty, write the block back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. For a write-through cache: Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. The Index Store Tag must be used to initialize the cache at power up.	Required
	S, T	Index Writeback Invalidate / Index Invalidate	Index		Required if S, T cache is implemented
0b001	All	Index Load Tag	Index	Read the tag for the cache block at the specified index into the *TagLo* and *TagHi* Coprocessor 0 registers. If the *DataLo* and *DataHi* registers are implemented, also read the data corresponding to the byte index into the *DataLo* and *DataHi* registers. This operation must not cause a Cache Error Exception. The granularity and alignment of the data read into the *DataLo* and *DataHi* registers is implementation-dependent, but is typically the result of an aligned access to the cache, ignoring the appropriate low-order bits of the byte index.	Recommended

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 5.20 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 *TagLo* and *TagHi* Coprocessor 0 registers. This operation must not cause a Cache Error Exception. This required encoding may be used by software to initialize the entire instruction or data caches by stepping through all valid indices. Doing so requires that the *TagLo* and *TagHi* registers associated with the cache be initialized first.	Required
0b011	All	Implementation Dependent	Unspecified	Available for implementation-dependent operation.	Optional
0b100	I, D	Hit Invalidate	Address	If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Required (Instruction Cache Encoding Only), Recommended otherwise
	S, T	Hit Invalidate	Address		Optional, if Hit_Invalidate_D is implemented, the S and T variants are recommended.
0b101	I	Fill	Address	Fill the cache from the specified address.	Recommended
	D	Hit Writeback Invalidate / Hit Invalidate	Address	For a write-back cache: If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Required
	S, T	Hit Writeback Invalidate / Hit Invalidate	Address		Required if S, T cache is implemented

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 5.20 Encoding of Bits [20:18] of the CACHE Instruction (Continued)

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Restrictions:

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

The operation of this instruction is **UNDEFINED** if the operation requires an address, and that address is uncacheable.

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

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

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

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

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

Availability and Compatibility:

This instruction has been reallocated an opcode in Release 6.

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

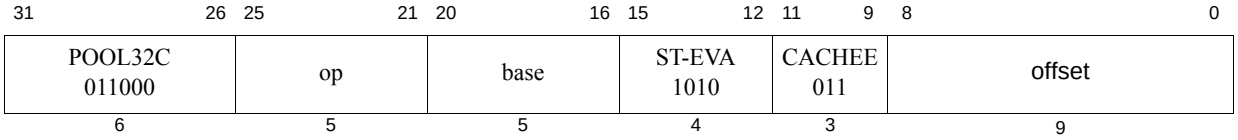
Programming Notes:

Release 6 architecture implements a 9-bit offset, whereas all release levels lower than Release 6 implement a 16-bit offset.

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

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


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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:

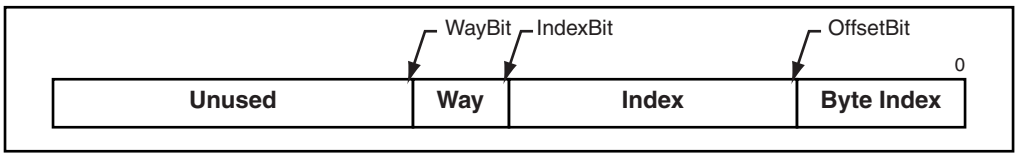
The 9-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

Table 5.21 Usage of Effective Address

Operation Requires an	Type of Cache	Usage of Effective Address
Address	Virtual	The effective address is used to address the cache. An address translation may or may not be performed on the effective address (with the possibility that a TLB Refill or TLB Invalid exception might occur)
Address	Physical	The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache

Index	N/A	The effective address is translated by the MMU to a physical address. It is implementation dependent whether the effective address or the translated physical address is used to index the cache. As such, a kseg0 address should always be used for cache operations that require an index. See the Programming Notes section below.
		
		Assuming that the total cache size in bytes is CS, the associativity is A, and the number of bytes per tag is BPT, the following calculations give the fields of the address which specify the way and the index:
		
		`OffsetBit ← Log2(BPT)`
		`IndexBit ← Log2(CS / A)`
		`WayBit ← IndexBit + Ceiling(Log2(A))`
		$Way \leftarrow Addr_{WayBit-1..IndexBit}$
		$Index \leftarrow Addr_{IndexBit-1..OffsetBit}$
		For a direct-mapped cache, the Way calculation is ignored and the Index value fully specifies the cache tag. This is shown symbolically in the figure below.

Figure 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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

operations (where the address is used to index the cache but need not match the cache tag) software should use unmapped addresses to avoid TLB exceptions. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS. This instruction never causes Execute-Inhibit nor Read-Inhibit exceptions.

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

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

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

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

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

Bits [22:21] of the instruction specify the cache on which to perform the operation, as follows:

Table 5.22 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

When implementing multiple level of caches and where the hardware maintains the smaller cache as a proper subset of a larger cache, it is recommended that the CACHEE instructions must first operate on the smaller, inner-level cache. For example, a Hit_Writeback _Invalidate operation targeting the Secondary cache, must 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.

When implementing multiple level of caches without the inclusion property, you must use SYNC instruction after the CACHEE instruction 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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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 the same 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 1.

Table 5.23 Encoding of Bits [20:18] of the CACHEE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b000	I	Index Invalidate	Index	Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.	Required
	D	Index Writeback Invalidate / Index Invalidate	Index	For a write-back cache: If the state of the cache block at the specified index is valid and dirty, write the block back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. For a write-through cache: Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. Note that Index Store Tag should be used to initialize the cache at power up.	Required
	S, T	Index Writeback Invalidate / Index Invalidate	Index		Required if S, T cache is implemented
0b001	All	Index Load Tag	Index	Read the tag for the cache block at the specified index into the *TagLo* and *TagHi* Coprocessor 0 registers. If the *DataLo* and *DataHi* registers are implemented, also read the data corresponding to the byte index into the *DataLo* and *DataHi* registers. This operation must not cause a Cache Error Exception. The granularity and alignment of the data read into the *DataLo* and *DataHi* registers is implementation-dependent, but is typically the result of an aligned access to the cache, ignoring the appropriate low-order bits of the byte index.	Recommended

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 5.23 Encoding of Bits [20:18] of the CACHEE 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 *TagLo* and *TagHi* Coprocessor 0 registers. This operation must not cause a Cache Error Exception. This required encoding may be used by software to initialize the entire instruction or data caches by stepping through all valid indices. Doing so requires that the *TagLo* and *TagHi* registers associated with the cache be initialized first.	Required
0b011	All	Implementation Dependent	Unspecified	Available for implementation-dependent operation.	Optional
0b100	I, D	Hit Invalidate	Address	If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Required (Instruction Cache Encoding Only), Recommended otherwise
	S, T	Hit Invalidate	Address		Optional, if Hit_Invalidate_D is implemented, the S and T variants are recommended.
0b101	I	Fill	Address	Fill the cache from the specified address.	Recommended
	D	Hit Writeback Invalidate / Hit Invalidate	Address	For a write-back cache: If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.	Required
	S, T	Hit Writeback Invalidate / Hit Invalidate	Address		Required if S, T cache is implemented

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 5.23 Encoding of Bits [20:18] of the CACHEE Instruction (Continued)

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Restrictions:

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

The operation of this instruction is **UNDEFINED** if the operation requires an address, and that address is uncacheable.

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

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

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

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

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

Operation:

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

Exceptions:

TLB Refill Exception.

TLB Invalid Exception

Coprocessor Unusable Exception

Reserved Instruction

Address Error Exception

Cache Error Exception

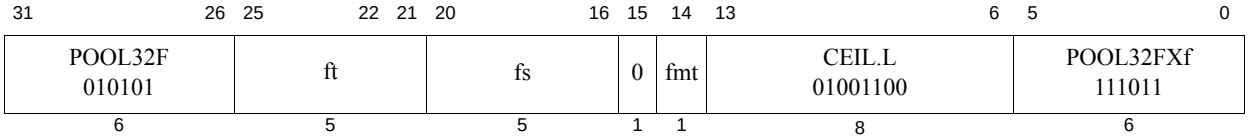
Bus Error Exception

Programming Notes:

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

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


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `CEIL.L.fmt`

`CEIL.L.S ft, fs` **microMIPS**

`CEIL.L.D ft, fs` **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] ← convert_and_round(FPR[fs])`

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

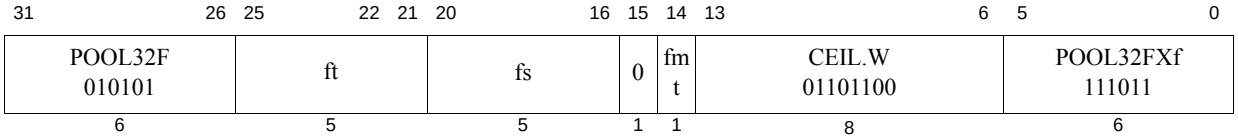
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `CEIL.W.fmt`

`CEIL.W.S ft, fs` **microMIPS**

`CEIL.W.D ft, fs` **microMIPS**

Purpose: Floating Point Ceiling Convert to Word Fixed Point

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

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

The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format and rounding toward +∞ (rounding mode 2). The result is placed in FPR *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, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}–1$, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed point. If the fields 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))
```

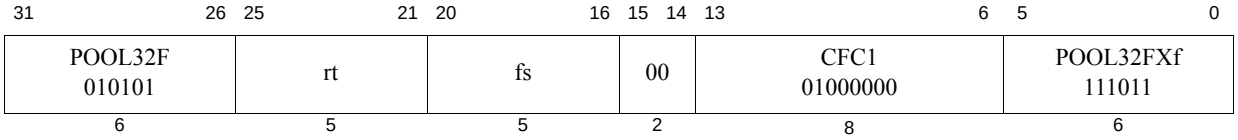
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Move Control Word From Floating Point

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

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

Copy the 32-bit word from FP (coprocessor 1) control register *fs* into GPR *rt*.

UFR (User FR change facility):

The UFR facility is removed in Release 6. Accessing the UFR and UNFR registers either cannot occur because Release 6 does not allow FIR_{UFRP} to be set, or is required to produce a Reserved Instruction Exception.

Restrictions:

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

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

Operation:

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

GPR[rt] ← temp
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Exceptions:

Coprocessor Unusable, Reserved Instruction

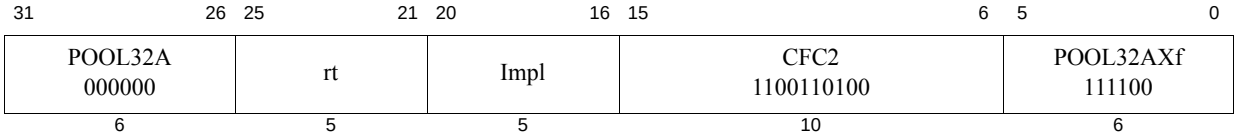
Historical Information:

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

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

MIPS32 Release 5 introduced the UFR and UNFR register aliases that allow user level access to $Status_{FR}$. Release 6 removes them.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← CP2CCR[Impl]`

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

Restrictions:

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

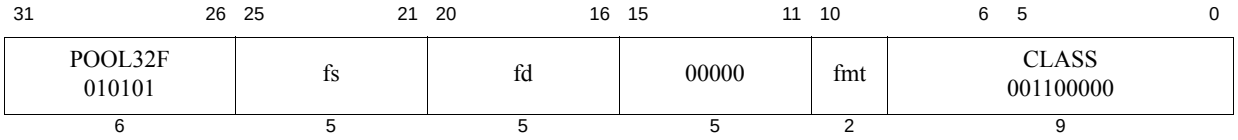
Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `CLASS.fmt`

`CLASS fd, fs, fmt` **microMIPS32 Release 6**

`CLASS.S fd,fs` **microMIPS32 Release 6**

`CLASS.D fd,fs` **microMIPS32 Release 6**

Purpose: Scalar Floating-Point Class Mask

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

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

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

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

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

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

The input operand is a scalar value in floating-point data format *fmt*. Bits beyond the width of *fmt* are ignored. The result is a 10-bit bitmask as described above, zero extended to 32-bits. Coprocessor register bits beyond 32-bits are UNPREDICTABLE (e.g. bits 32-63 are UNPREDICTABLE on a 64-bit FPU, while bits 32-128 bits are UNPREDICTABLE if the processor supports MSA).

Restrictions:

No data-dependent exceptions are possible.

Availability and Compatibility:

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

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

Operation:

```
CLASS.fmt
if not IsCoprocessorEnabled(1)
   then SignalException(CoprocessorUnusable, 1) end if
if not IsFloatingPointImplemented(fmt))
   then SignalException(ReservedInstruction) end if
if fmt=D and FIR.D=0
   then SignalFPException(UnimplementedOperation) end if

fin ← ValueFPR(fs,fmt)
masktmp ← ClassFP(fin, fmt)
StoreFPR (fd, fmt, ftmp )
/* end of instruction */
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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

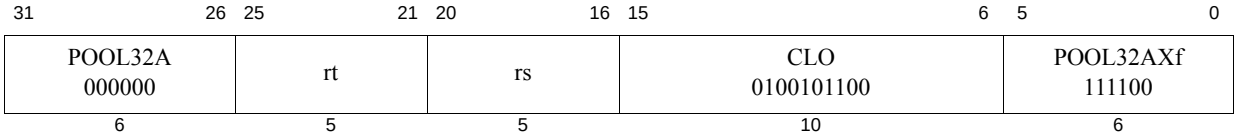
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `CLO rt, rs` **microMIPS**

Purpose: Count Leading Ones in Word

To count the number of leading ones in a word.

Description: `GPR[rt] ← 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:

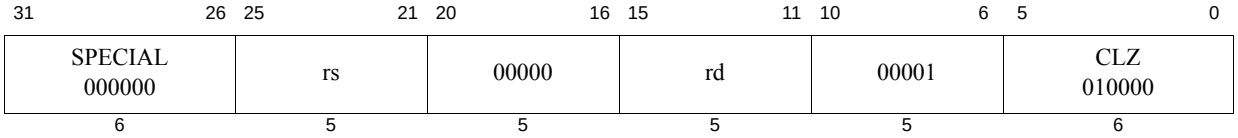
Operation:

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

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `CLZ rt, rs` **microMIPS**

Purpose: Count Leading Zeros in Word

Count the number of leading zeros in a word.

Description: `GPR[rt] ← 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:

Availability and Compatibility:

This instruction has been reallocated an opcode in Release 6.

Operation:

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

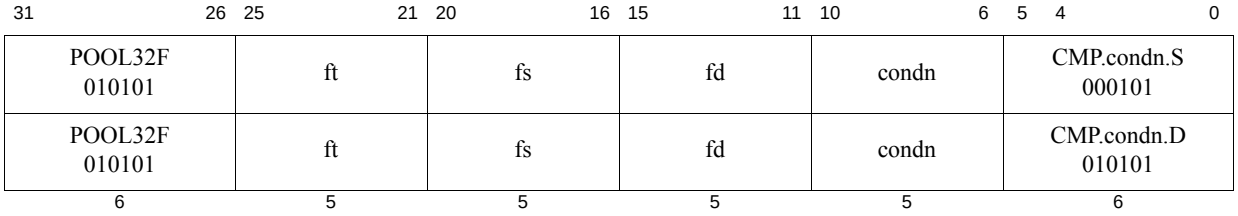
Exceptions:

None

Programming Notes:

Release 6 sets the ‘rt’ field to a value of 00000.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



31–26	25–21	20–16	15–11	10–6	5–0
POOL32F 010101	ft	fs	fd	condn	CMP.condn.S 000101
POOL32F 010101	ft	fs	fd	condn	CMP.condn.D 010101
6	5	5	5	5	6

Format: `CMP.condn.fmt`

`CMP.condn.S fd, fs, ft` **microMIPS32 Release 6**

`CMP.condn.D fd, fs, ft` **microMIPS32 Release 6**

Purpose: Floating Point Compare setting Mask

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

Description: `FPR[fd] ← FPR[fs] compare_cond FPR[ft]`

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

The comparison is exact and neither overflows nor underflows.

If the comparison specified by the *condn* field of the instruction is true for the operand values, the result is true; otherwise, the result is false. If no exception is taken, the result is written into FPR *fd*; true is all 1s and false is all 0s, repeated the operand width of *fmt*. All other bits beyond the operand width *fmt* are UNPREDICTABLE. E.g. a 32-bit single precision comparison writes a mask of 32 0s or 1s into bits 0 to 31 of FPR *fd*, and makes bits 32 to 63 to UNPREDICTABLE, if a 64-but FPU without MSA, or bits 32 to 127 if MSA is present.

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

All other encodings, that is all other values of *fmt,* are reserved in Release 6, and produce a Reserved Instruction Exception. The encodings corresponding to MIPS32 Release 5 C.cond.S and C.cond.D are so reserved.

The *condn* field of the instruction specifies the nature of the comparison: equals, less than, and so on, unordered or ordered, signalling or quiet, as specified in Table 5.24 “Comparing CMP.condn.fmt, IEEE 754-2008, C.cond.fmt, and MSA FP compares” on page 166.

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

All encodings of the *condn* field that are not specified. For example, items shaded in Table 5.24, are reserved in Release 6 and produce a Reserved Instruction exception.

If one of the values is an SNaN, or if a signalling comparison is specified (currently, $cond_3$ is set) and at least one of the values is a QNaN, an Invalid Operation condition is raised and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written and an Invalid Operation exception is taken immediately. Otherwise, the mask result is written into FPR *fd.*

There are four mutually exclusive ordering relations for comparing floating point values; one relation is always true and the others are false. The familiar relations are *greater than*, *less than*, and *equal*. In addition, the IEEE floating point standard defines the relation *unordered,* which is true when at least one operand value is NaN; NaN compares

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

unordered with everything, including itself. Comparisons ignore the sign of zero, so +0 equals -0.

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

The predicates implemented are described in Table 5.24 “Comparing CMP.condn.fmt, IEEE 754-2008, C.cond.fmt, and MSA FP compares” on page 166. Not all of the 16 IEEE predicates are implemented directly by hardware. For the directed comparisons (LT, LE, GT, GE) the missing predicates can be obtained by reversing the FPR register operands *ft* and *fs*. For example, the hardware implements the “Ordered Less Than” predicate LT(fs,ft); reversing the operands LT(ft,fs) produces the dual predicate “Unordered or Greater Than or Equal” UGE(fs,ft). Table 5.24 shows these mappings. Reversing inputs is ineffective for the symmetric predicates such as EQ; Release 6 implements these negative predicates directly, so that all mask values can be generated in a single instruction.

In addition to showing the comparison predicates, instruction encodings, instruction mnemonics and longer instruction names for CMP.condn.fmt, Table 5.24 provides comparisons to (1) the MIPS32 Release 5C.cond.fmt, and (2) the (MSA) MIPS SIMD Architecture packed vector floating point comparison instructions. CMP.condn.fmt provides exactly the same comparisons for FPU scalar values that MSA provides for packed vectors, with similar mnemonics. CMP.condn.fmt provides a superset of the MIPS32 Release 5 C.cond.fmt comparisons.

The official Release 6 mnemonics for comparisons have slightly changed. For example, SULT, Signalling Unordered or Less Than. Table 5.24 indicates mnemonics that implement the same predicates.

In addition, Table 5.24 shows the corresponding IEEE 754-2008 comparison operations.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 5.24 Comparing CMP.condn.fmt, IEEE 754-2008, C.cond.fmt, and MSA FP compares

Shaded entries in the table are unimplemented, and reserved.

Instruction Encodings

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

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved

Table 5.24 Comparing CMP.condn.fmt, IEEE 754-2008, C.cond.fmt, and MSA FP compares (Continued)

Shaded entries in the table are unimplemented, and reserved.

Instruction Encodings

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

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved

Restrictions:

Operation:

```
if SNaN(ValueFPR(fs, fmt)) or SNaN(ValueFPR(ft, fmt)) or
   QNaN(ValueFPR(fs, fmt)) or QNaN(ValueFPR(ft, fmt))
then
   less ← false
   equal ← false
   unordered ← true
   if (SNaN(ValueFPR(fs,fmt)) or SNaN(ValueFPR(ft,fmt))) or
      (cond3 and (QNaN(ValueFPR(fs,fmt)) or QNaN(ValueFPR(ft,fmt)))) then
         SignalException(InvalidOperation)
   endif
   else
      less ← ValueFPR(fs, fmt) <fmt ValueFPR(ft, fmt)
      equal ← ValueFPR(fs, fmt) =fmt ValueFPR(ft, fmt)
      unordered ← false
   endif
   condition ← cond4 xor (
          (cond2 and less)
          or (cond1 and equal)
          or (cond0 and unordered) )
   StoreFPR (fd, fmt, ExtendBit.fmt(condition))
endif
```

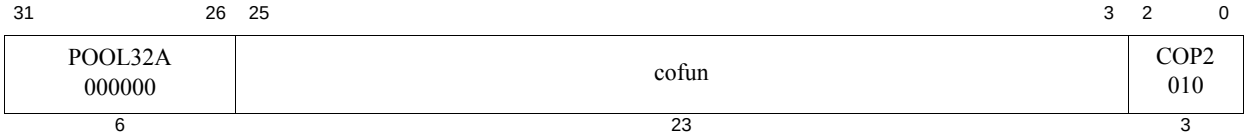
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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:

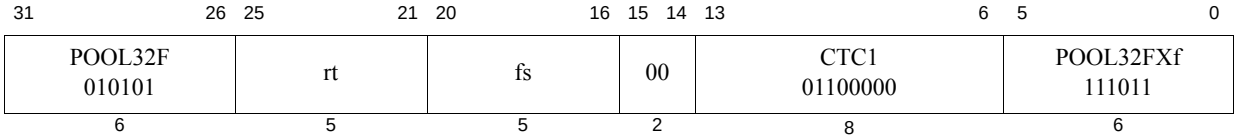
Operation:

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

Exceptions:

Coprocessor Unusable
Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Move Control Word to Floating Point

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

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

Copy the low word from GPR *rt* into the FP (coprocessor 1) control register indicated by *fs*.

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

UFR (User FR change facility):

The UFR facility is removed in Release 6. Accessing the UFR and UNFR registers either cannot occur because Release 6 does not allow FIR_{UFRP} to be set, or is required to produce a Reserved Instruction Exception.

Restrictions:

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

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

Operation:

```
temp ← GPR[rt]31..0
if (fs = 1 or fs = 4) then
   /* clear UFR or UNFR(CP1 Register 1)*/
   if Config2.AR>=2 SignalException(RI) /* MIPS Release 6 traps */ endif
   if not (rt = 0 and FIRUFRP) then UNPREDICTABLE /*end of instruction*/ endif
   if not Config5UFR then signalException(RI) endif?
   if fs = 1 then StatusFR ← 0
   elseif fs = 4 then StatusFR ← 1
   else /* cannot happen */
elseif fs = 25 then /* FCCR */
   if temp31..8 ... 024 then
      UNPREDICTABLE
   else
      FCSR ← temp7..1 || FCSR24 || temp0 || FCSR22..0
   endif
elseif fs = 26 then /* FEXR */
   if temp31..18 ... 0 or temp11..7 ... 0 or temp2..0 ... 0then
      UNPREDICTABLE
   else
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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

CheckFPException()
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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

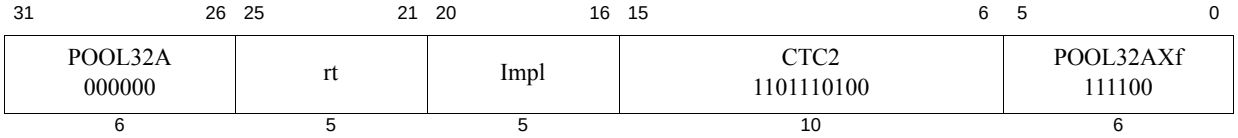
Historical Information:

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

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

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← 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.

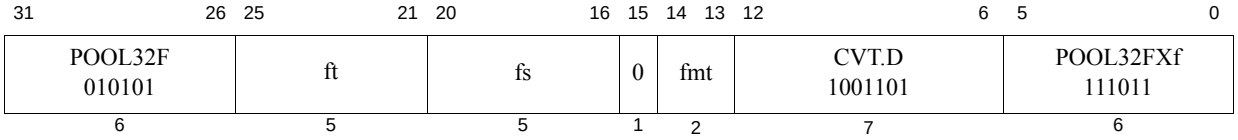
Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `CVT.D.fmt`

`CVT.D.S ft, fs` **microMIPS**

`CVT.D.W ft, fs` **microMIPS**

`CVT.D.L ft, fs` **microMIPS**

Purpose: Floating Point Convert to Double Floating Point

To convert an FP or fixed point value to double FP.

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

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

Restrictions:

The fields *fs* and *ft* must specify valid FPRs, *fs* for type *fmt* and *ft* for double floating point. If the fields are not valid, the result is **UNPREDICTABLE**.

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

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

Operation:

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

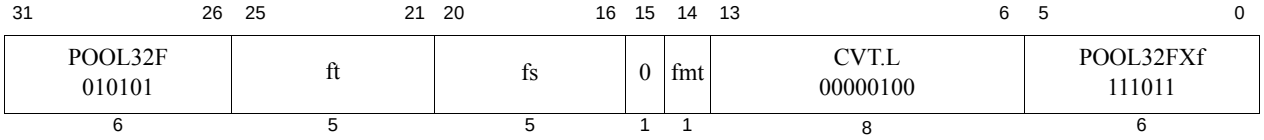
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `CVT.L.fmt`

`CVT.L.S ft, fs` **microMIPS**

`CVT.L.D ft, fs` **microMIPS**

Purpose: Floating Point Convert to Long Fixed Point

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

Description: `FPR[ft] ← 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 the fields 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))
```

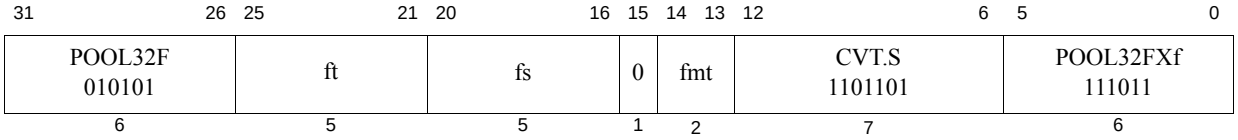
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact,

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `CVT.S.fmt`

`CVT.S.D ft, fs` **microMIPS**

`CVT.S.W ft, fs` **microMIPS**

`CVT.S.L ft, fs` **microMIPS**

Purpose: Floating Point Convert to Single Floating Point

To convert an FP or fixed point value to single FP.

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

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

Restrictions:

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

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

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

Operation:

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

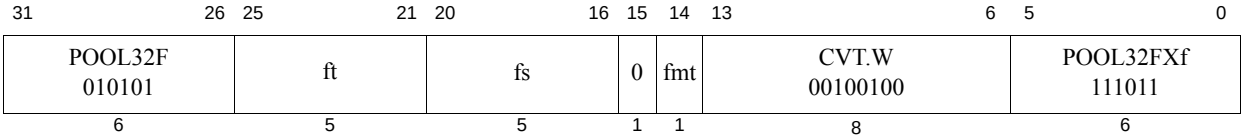
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow, Underflow

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `CVT.W.fmt`

`CVT.W.S ft, fs` **microMIPS**

`CVT.W.D ft, fs` **microMIPS**

Purpose: Floating Point Convert to Word Fixed Point

To convert an FP value to 32-bit fixed point.

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

The value in FPR *fs,* in format *fmt,* is converted to a value in 32-bit word fixed point format and rounded according to the current rounding mode in *FCSR*. The result is placed in FPR *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, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *ft*.

Restrictions:

The fields *fs* and *ft* must specify valid FPRs: *fs* for type *fmt* and *ft* for word fixed point. If the fields 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))
```

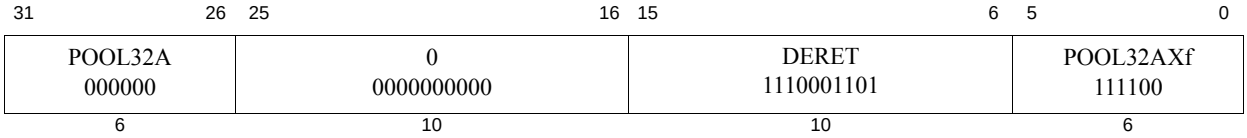
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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.
Pre-Release 6: The operation of the processor is **UNDEFINED** if a DERET is executed in the delay slot of a branch or jump instruction.This restriction does not apply in Release 6.

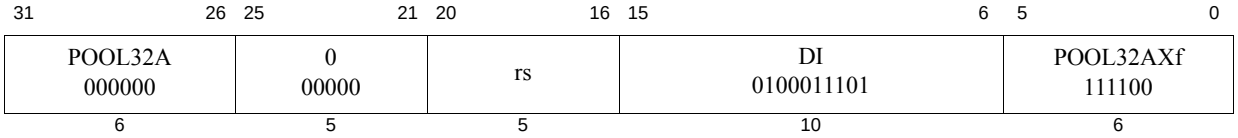
Operation:

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

Exceptions:

Coprocessor Unusable Exception
Reserved Instruction Exception

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `DI` **microMIPS**

`DI rs` **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] ← Status; Status`$_{IE}$ `← 0`

The current value of the *Status* register is 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] ← data
Status_IE ← 0
```

Exceptions:

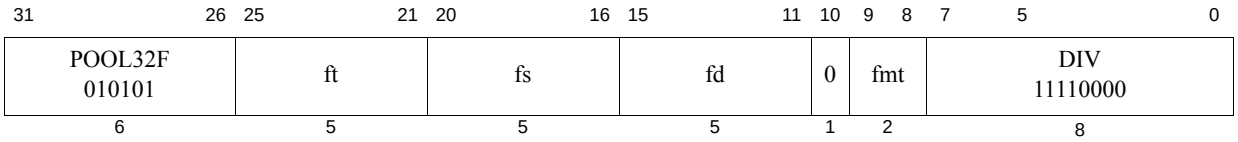
Coprocessor Unusable
Reserved Instruction (Release 1 implementations)

Programming Notes:

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

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `DIV.fmt`

`DIV.S fd, fs, ft` **microMIPS**

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

Purpose: Floating Point Divide

To divide FP values.

Description: `FPR[fd] ← FPR[fs] / FPR[ft]`

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

Restrictions:

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

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

Operation:

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

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

31 26	25 21	20 16	15 11	10	9 0
POOL32A 000000	rt	rs	rd	0	DIV 0100011000
POOL32A 000000	rs	rt	rd	0	MOD 0101011000
POOL32A 000000	rs	rt	rd	0	DIVU 0110011000
POOL32A 000000	rs	rt	rd	0	MODU 0111011000
6	5	5	5	1	10

Format: `DIV MOD DIVU MODU`

`DIV rd,rs,rt` **microMIPS32 Release 6**

`MOD rd,rs,rt` **microMIPS32 Release 6**

`DIVU rd,rs,rt` **microMIPS32 Release 6**

`MODU rd,rs,rt` **microMIPS32 Release 6**

micromicromicromicro

Purpose: Divide Integers (with result to GPR)

DIV: Divide Words Signed
MOD: Modulo Words Signed
DIVU: Divide Words Signed
MODU: Modulo Words Signed

Description:

```
DIV:   GPR[rd] ← ( divide.signed( GPR[rs] div GPR[rt] )
MOD:   GPR[rd] ← ( modulo.signed( GPR[rs] mod GPR[rt] )
DIVU:  GPR[rd] ← ( divide.unsigned( GPR[rs] div GPR[rt] )
MODU:  GPR[rd] ← ( modulo.unsigned( GPR[rs] mod GPR[rt] )
```

The Release 6 divide and modulo instructions divide the operands in GPR `rs` and GPR `rt`, and place the quotient or remainder in GPR rd.

For each of the div/mod operator pairs DIV/M OD, DIVU/MODU, the results satisfy the equation `(A div B)*B + (A mod B) = A`, where `(A mod B)` has same sign as the dividend `A`, and `abs(A mod B) < abs(B)`. This equation uniquely defines the results.

NOTE: if the divisor B=0, this equation cannot be satisfied, and the result is UNPREDICTABLE. This is commonly called “truncated division”.

DIV performs a signed 32-bit integer division, and places the 32-bit quotient result in the destination register.

MOD performs a signed 32-bit integer division, and places the 32-bit remainder result in the destination register. The remainder result has the same sign as the dividend.

DIVU performs an unsigned 32-bit integer division, and places the 32-bit quotient result in the destination register.

MODU performs an unsigned 32-bit integer division, and places the 32-bit remainder result in the destination register.

Restrictions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Availability and Compatibility:

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

Release 6 divide instructions have the same opcode mnemonic as the pre-Release 6 divide instructions (DIV, DIVU). The instruction encodings are different, as are the instruction semantics: the Release 6 instruction produces only the quotient, whereas the pre-Release 6 instruction produces quotient and remainder in HI/LO registers respectively, and separate modulo instructions are required to obtain the remainder.

The assembly syntax distinguishes the Release 6 from the pre-Release 6 divide instructions. For example, Release 6 "`DIV rd,rs,rt`" specifies 3 register operands, versus pre-Release 6 "`DIV rs,rt`", which has only two register arguments, with the HI/LO registers implied. Some assemblers accept the pseudo-instruction syntax "`DIV rd,rs,rt`" and expand it to do "`DIV rs,rt;MFHI rd`". Phrases such as "DIV with GPR output" and "DIV with HI/LO output" may be used when disambiguation is necessary.

Pre-Release 6 divide instructions that produce quotient and remainder in the HI/LO registers produce a Reserved Instruction Exception on Release 6. In the future, the instruction encoding may be reused for other instructions.

Programming Notes:

Because the divide and modulo instructions are defined to not trap if dividing by zero, it is safe to emit code that checks for zero-divide after the divide or modulo instruction.

Operation

```
DIV, MOD: if NotWordValue(GPR[rs]) then UNPREDICTABLE endif
DIV, MOD: if NotWordValue(GPR[rt]) then UNPREDICTABLE endif
DIVU, MODU: if not(zero_or_sign_extended.32(GPR[rs])) then UNPREDICTABLE endif
DIVU, MODU: if not(zero_or_sign_extended.32(GPR[rt])) then UNPREDICTABLE endif
DIV, MOD:
  s1 ← signed_word(GPR[rs])
  s2 ← signed_word(GPR[rt])
DIVU, MODU:
  s1 ← unsigned_word(GPR[rs])
  s2 ← unsigned_word(GPR[rt])

DIV, DIVU:
  quotient ← s1 div s2
MOD, MODU:
  remainder ← s1 mod s2

DIV:   GPR[rd] ←  quotient
MOD:   GPR[rd] ←  remainder
DIVU:  GPR[rd] ←  quotient
MODU:  GPR[rd] ←  remainder
/* end of instruction */
```

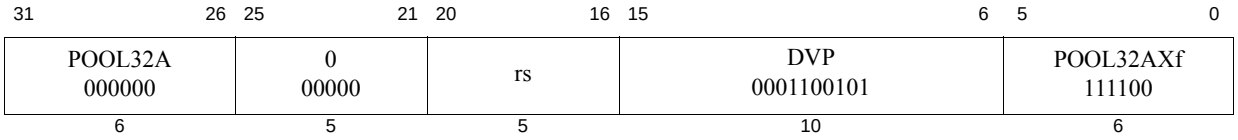
Exceptions:

None[1]

1. No arithmetic exception occurs under any circumstances. Division by zero produces an UNPREDICTABLE result.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `DVP` **microMIPS Release 6**

`DVP rs` **microMIPS Release 6**

Purpose: Disable Virtual Processor

To disable all virtual processors in a physical core other than the virtual processor that issued the instruction.

Description: `GPR[rs] ← VPControl ; VPControl`$_{DIS}$ `← 1`

Disabling a virtual processor means that instruction fetch is terminated, and all outstanding instructions for the affected virtual processor(s) must be complete before the DVP itself is allowed to retire. Any outstanding events such as hardware instruction or data prefetch, or page-table walks must also be terminated.

The DVP instruction has implicit SYNC(*stype*=0) semantics but with respect to the other virtual processors in the physical core.

After all other virtual processors have been disabled, $VPControl_{DIS}$ is set. Prior to modification and if *rs* is non-zero, *VPControl* is written to GPR[*rs*]. If DVP is specified without *rs*, then *rs* must be 0.

DVP may also take effect on a virtual processor that has executed a WAIT or a PAUSE instruction. If a virtual processor has executed a WAIT instruction, then it cannot resume execution on an interrupt until an EVP has been executed. If the EVP is executed before the interrupt arrives, then the virtual processor resumes in a state as if the DVP had not been executed, that is, it waits for the interrupt.

If a virtual processor has executed a PAUSE instruction, then it cannot resume execution until an EVP has been executed, even if LLbit is cleared. If an EVP is executed before the LLbit is cleared, then the virtual processor resumes in a state as if the DVP has not been executed, that is, it waits for the LLbit to clear.

The execution of a DVP must be followed by the execution of an EVP. The execution of an EVP causes execution to resume immediately—where applicable—on all other virtual processors, as if the DVP had not been executed. The execution is completely restorable after the EVP. If an event occurs in between the DVP and EVP that renders state of the virtual processor UNPREDICTABLE (such as power-gating), then the effect of EVP is UNPREDICTABLE.

DVP may only take effect if $VPControl_{DIS}$=0. Otherwise it is treated as a NOP instruction.

If a virtual processor is disabled due to a DVP, then interrupts are also disabled for the virtual processor, that is, logically $Status_{IE}$=0. $Status_{IE}$ for the target virtual processors though is not cleared though as software cannot access state on the virtual processors that have been disabled. The virtual processor which executes the DVP however continues to be interruptible.

In an implementation, the ability of a virtual processor to execute instructions may also be under control external to the physical core which contains the virtual processor. If disabled by DVP, a virtual processor must not resume fetch in response to the assertion of this external signal to enable fetch. Conversely, if fetch is disabled by such external control, then execution of EVP will not cause fetch to resume at a target virtual processor for which the control is deasserted.

This instruction never executes speculatively. It must be the oldest unretired instruction to take effect.

This instruction is only available in Release 6 implementations. For implementations that do not support multi-threading ($Config5_{VP}$=0), this instruction must be treated as a NOP instruction.

Restrictions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

In implementations prior to Release 6 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

The pseudo-code below assumes that the DVP is executed by virtual processor 0, while the target virtual processor is numbered ‘n’, where n is each of all remaining virtual processors.

```
            if (VPControl_DIS = 0)

            // Pseudo-code in italics provides recommended action wrt other VPs
            disable_fetch(VPn) {
                if PAUSE(VPn) retires prior or at disable event
                then VPn execution is not resumed if LLbit is cleared prior to EVP
            }
            disable_interrupt(VPn) {
                if WAIT(VPn) retires prior or at disable event
                then interrupts are ignored by VPn until EVP
            }
            // DVP0 not retired until instructions for VPn completed
            while (VPn outstanding instruction)
                DVP0 unretired
            endwhile

            endif

data ← VPControl
GPR[rs] ← data
VPControl_DIS ← 1
```

Exceptions:

Coprocessor Unusable
Reserved Instruction (pre-Release 6 implementations)

Programming Notes:

DVP may disable execution in the target virtual processor regardless of the operating mode - kernel, supervisor, user. Kernel software may also be in a critical region, or in a high-priority interrupt handler when the disable occurs. Since the instruction is itself privileged, such events are considered acceptable.

Before executing an EVP in a DVP/EVP pair, software should first read $\texttt{VPControl}_{\texttt{DIS}}$, returned by DVP, to determine whether the virtual processors are already disabled. If so, the DVP/EVP sequence should be abandoned. This step allows software to safely nest DVP/EVP pairs.

Privileged software may use DVP/EVP to disable virtual processors on a core, such as for the purpose of doing a cache flush without interference from other processes in a system with multiple virtual processors or physical cores.

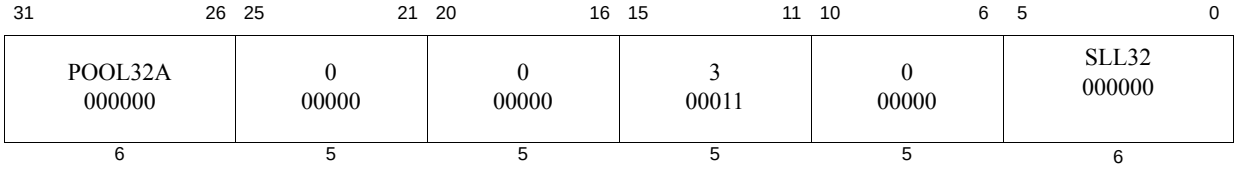
DVP (and EVP) may be used in other cases such as for power-savings or changing state that is applicable to all virtual processors in a core, such as virtual processor scheduling priority, as described below :

```
ll t0 0(a0)
dvp    // disable all other virtual processors
pause  // wait for LLbit to clear
evp    // enable all othe virtual processors
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
ll t0 0(a0)
dvp    // disable all other virtual processors
<change core-wide state>
evp    // enable all othe virtual processors
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `EHB` **microMIPS**

Purpose: Execution Hazard Barrier

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

Description:

EHB is 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. 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()
```

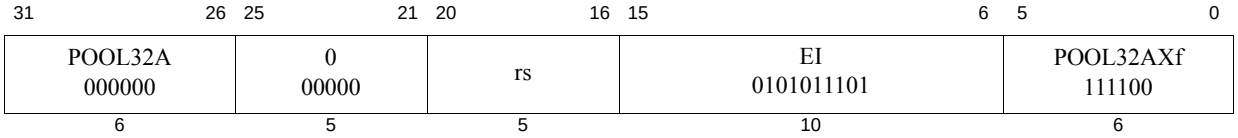
Exceptions:

None

Programming Notes:

In 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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `EI` **microMIPS**

`EI rs` **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] ← Status; Status`$_{IE}$ `← 1`

The current value of the *Status* register is 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] ← data
Status_IE ← 1
```

Exceptions:

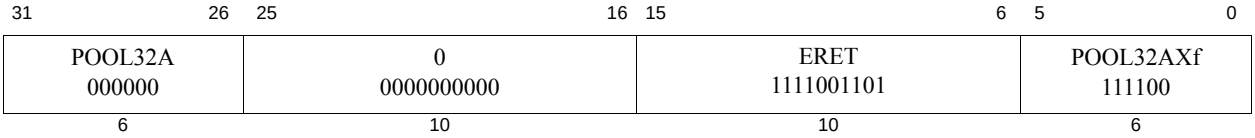
Coprocessor Unusable
Reserved Instruction (Release 1 implementations)

Programming Notes:

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

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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 (that is, it has no delay slot).

Restrictions:

Pre-Release 6: The operation of the processor is **UNDEFINED** if an ERET is executed in the delay slot of a branch or jump instruction. This restriction does not apply in Release 6.

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

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

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

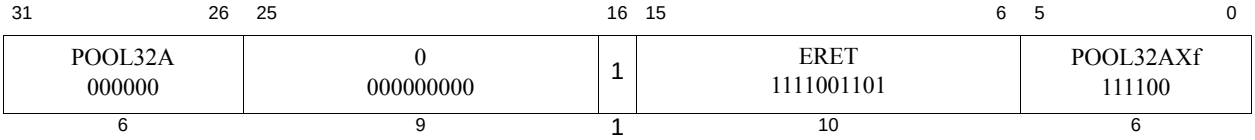
Operation:

```
if Status_ERL = 1 then
   temp ← ErrorEPC
   Status_ERL ← 0
else
   temp ← EPC
   Status_EXL ← 0
   if (ArchitectureRevision ≥ 2) and (SRSCtl_HSS > 0) and (Status_BEV = 0) then
      SRSCtl_CSS ← SRSCtl_PSS
   endif
endif
if IsMIPS16Implemented() | (Config3_ISA > 0) then
   PC ← temp_31..1 || 0
   ISAMode ← temp_0
else
   PC ← temp
endif
LLbit ← 0
ClearHazards()
```

Exceptions:

Coprocessor Unusable Exception

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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 must 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, because 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:

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

Operation:

```
if Status_ERL = 1 then
   temp ← ErrorEPC
   Status_ERL ← 0
else
   temp ← EPC
   Status_EXL ← 0
   if (ArchitectureRevision ≥ 2) and (SRSCtl_HSS > 0) and (Status_BEV = 0) then
      SRSCtl_CSS ← SRSCtl_PSS
   endif
endif
if IsMIPS16Implemented() | (Config3_ISA > 0) then
   PC ← temp_31..1 || 0
   ISAMode ← temp_0
else
```

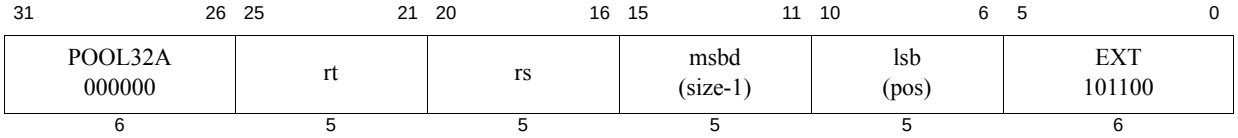

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
    PC ← temp
endif
ClearHazards()
```

Exceptions:

Coprocessor Unusable Exception

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `EXT rt, rs, pos, size` **microMIPS**

Purpose: Extract Bit Field

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

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

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

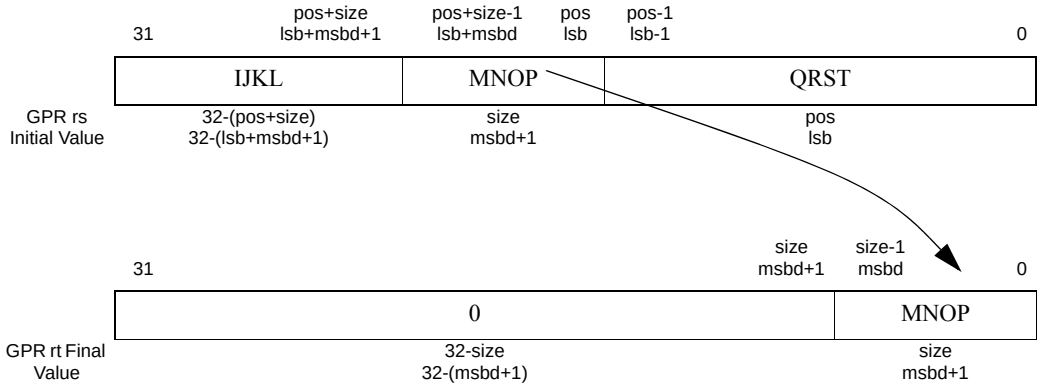
```
msbd ← size-1
lsb ← pos
```

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

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

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

Figure 5.4 Operation of the EXT Instruction



Restrictions:

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

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

Operation:

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

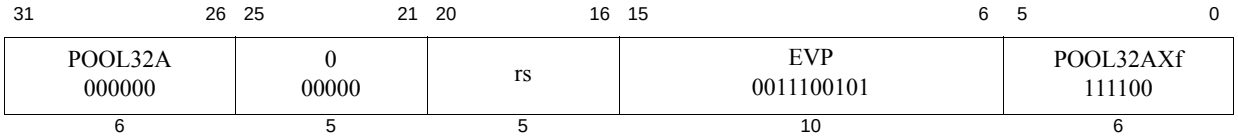

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
GPR[rt] ← temp
```

Exceptions:

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `EVP` **microMIPS Release 6**

`EVP rs` **microMIPS Release 6**

Purpose: Enable Virtual Processor

To enable all virtual processors in a physical core other than the virtual processor that issued the instruction.

Description: `GPR[rs] ← VPControl ; VPControl`$_{DIS}$ `← 0`

Enabling a virtual processor means that instruction fetch is resumed.

After all other virtual processors have been enabled, `VPControl`$_{DIS}$ is cleared. Prior to modification, if *rs* is non-zero, `VPControl` is written to GPR[*rs*]. If EVP is specified without *rs*, then *rs* must be 0.

See the DVP instruction to learn about the behavior of EVP in the context of WAIT/PAUSE/external-control ("DVP" on page 183).

The execution of a DVP must be followed by the execution of an EVP. The execution of an EVP causes execution to resume immediately, *where applicable*, on all other virtual processors, as if the DVP had not been executed, that is, execution is completely restorable after the EVP. On the other hand, if an event occurs in between the DVP and EVP that renders state of the virtual processor UNPREDICTABLE (such as power-gating), then the effect of EVP is UNPREDICTABLE.

EVP may only take effect if *VPControl*$_{DIS}$=1. Otherwise it is treated as a NOP

This instruction never executes speculatively. It must be the oldest unretired instruction to take effect.

This instruction is only available in Release 6 implementations. For implementations that do not support multi-threading (*Config5*$_{VP}$=0), this instruction must be treated as a NOP instruction.

Restrictions:

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

In implementations prior to Release 6 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

The pseudo-code below assumes that the EVP is executed by virtual processor 0, while the target virtual processor is numbered 'n', where n is each of all remaining virtual processors.

```
            if (VPControl_DIS = 1)

// Pseudo-code in italics provides recommended action wrt other VPs
enable_fetch(VPn) {
    if PAUSE(VPn) retires prior or at disable event
    then VPn execution is not resumed if LLbit is cleared prior to EVP
}
enable_interrupt(VPn) {
    if WAIT(VPn) retires prior or at disable event
    then interrupts are ignored by VPn until EVP
}
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
            endif

    data ← VPControl
    GPR[rs] ← data
    VPControl_DIS ← 0
```

Exceptions:

Coprocessor Unusable
Reserved Instruction (pre-Release 6 implementations)

Programming Notes:

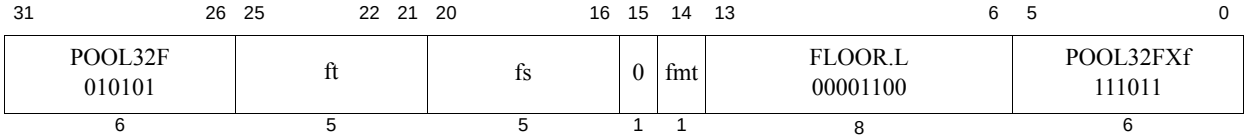
Before executing an EVP in a DVP/EVP pair, software should first read $\texttt{VPControl}_{\texttt{DIS}}$, returned by DVP, to determine whether the virtual processors are already disabled. If so, the DVP/EVP sequence should be abandoned. This step allows software to safely nest DVP/EVP pairs.

Privileged software may use DVP/EVP to disable virtual processors on a core, such as for the purpose of doing a cache flush without interference from other processes in a system with multiple virtual processors or physical cores.

DVP (and EVP) may be used in other cases such as for power-savings or changing state that is applicable to all virtual processors in a core, such as virtual processor scheduling priority, as described below:

```
ll t0 0(a0)
dvp    // disable all other virtual processors
pause  // wait for LLbit to clear
evp    // enable all othe virtual processors


ll t0 0(a0)
dvp    // disable all other virtual processors
<change core-wide state>
evp    // enable all othe virtual processors
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `FLOOR.L.fmt`

`FLOOR.L.S ft, fs` **microMIPS**

`FLOOR.L.D ft, fs` **microMIPS**

Purpose: Floating Point Floor Convert to Long Fixed Point

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

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

The value in FPR *fs*, in format *fmt*, is converted to a value in 64-bit long fixed point format and rounded toward ≥ (rounding mode 3). The result is placed in FPR *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 *ft* for long fixed point. If the fields 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))
```

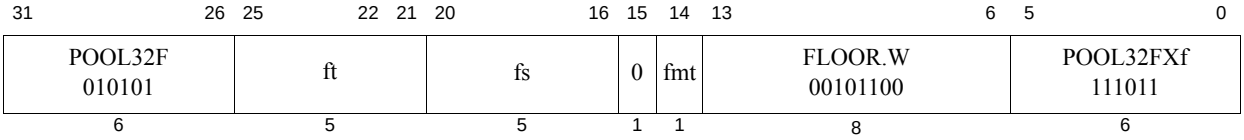
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `FLOOR.W.fmt`

`FLOOR.W.S ft, fs` **microMIPS**

`FLOOR.W.D ft, fs` **microMIPS**

Purpose: Floating Point Floor Convert to Word Fixed Point

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

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

The value in FPR *fs,* in format *fmt,* is converted to a value in 32-bit word fixed point format and rounded toward –≥ (rounding mode 3). The result is placed in FPR *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, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}–1$, is written to *ft*.

Restrictions:

The fields *fs* and *ft* must specify valid FPRs: *fs* for type *fmt* and *ft* for word fixed point. If the fields 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))
```

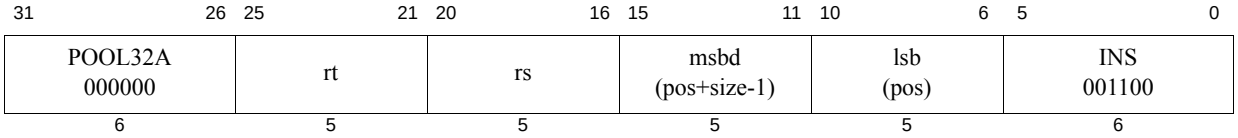
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `INS rt, rs, pos, size` **microMIPS**

Purpose: Insert Bit Field

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

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

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

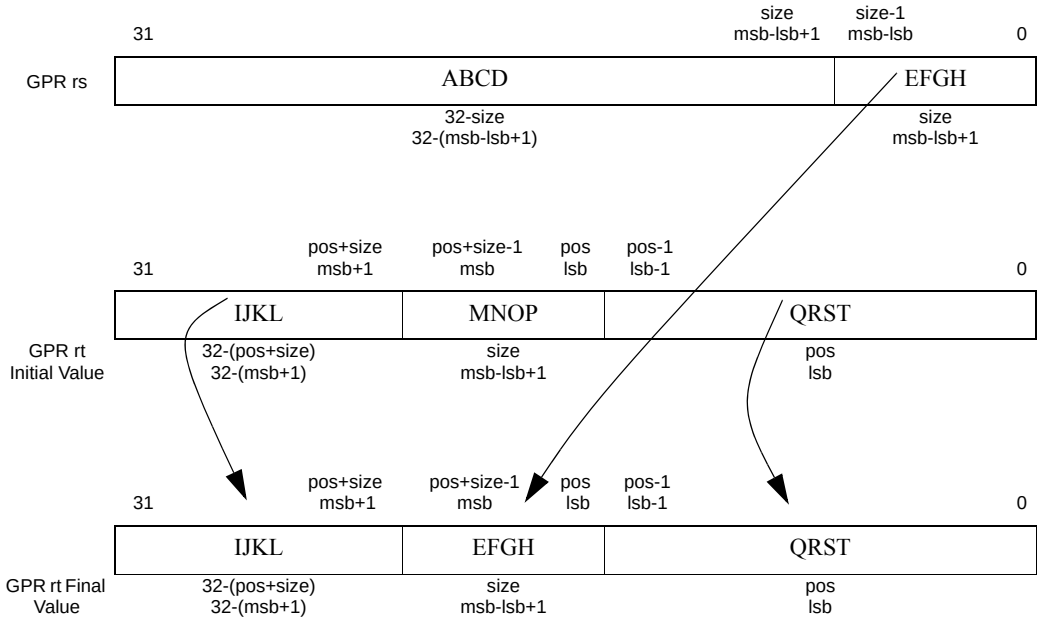
```
msb ← pos+size-1
lsb ← pos
```

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

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

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

Figure 5.5 Operation of the INS Instruction



Restrictions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

tion.

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

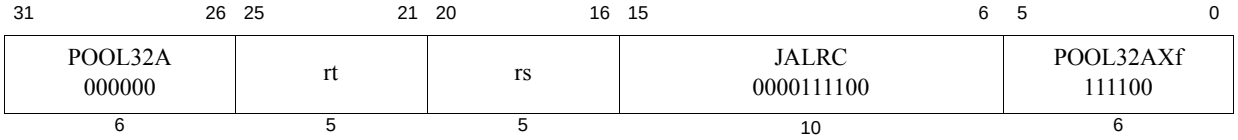
Operation:

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

Exceptions:

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `JALRC rs (rt = 31 implied)` **microMIPS Release 6**
`JALRC rt, rs` **microMIPS Release 6**

Purpose: Jump and Link Register Compact

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

Description: `GPR[rt] ← return_addr, PC ← GPR[rs]`

For processors that do not implement the MIPS ISA:

- Jump to the effective target address in GPR *rs*. Bit 0 of GPR *rs* is interpreted as the target ISA Mode: if this bit is 0, signal an Address Error exception when the target instruction is fetched because this target ISA Mode is not supported. Otherwise, set bit 0 of the target address to zero, and fetch the instruction.

For processors that do implement the MIPS ISA:

- Jump to the effective target address in GPR *rs*. Set the ISA Mode bit to the value in GPR *rs* bit 0. Set bit 0 of the target address to zero. If the target ISA Mode bit is 0 and the target address is not 4-byte aligned, an Address Error exception will occur when the target instruction is fetched.

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.

Compact jumps do not have delay slots. The instruction after the jump is NOT executed when the jump is executed.

Availability and Compatibility:

Release 6 maps JR and JR.HB to JALRC and JALRC.HB with rt = 0:

Release 6 assemblers should accept the JR and JR.HB mnemonics, mapping them to the Release 6 instruction encodings.

Restrictions:

Restrictions Related to Multiple Instruction Sets: This instruction can change the active instruction set, if more than one instruction set is implemented.

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 MIPS and if the ISAMode bit of the target is MIPS (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 MIPS ISA, if the intended target ISAMode is MIPS (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

Operation:

```
temp ← GPR[rs]
   GPR[rt] ← PC + 4
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
if Config3ISA = 1 then
       PC ← temp
    else
       PC ← tempGPRLEN-1..1 || 0
       ISAMode ← temp0
    endif
```

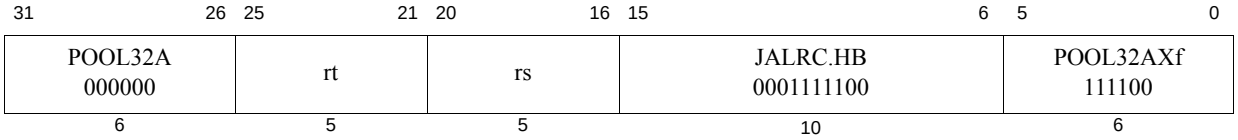
Exceptions:

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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `JALRC.HB rs (rt = 31 implied)` **microMIPS Release 6**

`JALRC.HB rt, rs` **microMIPS Release 6**

Purpose: Jump and Link Register Compact 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] ← return_addr, PC ← GPR[rs], clear execution and instruction hazards`

For processors that do not implement the MIPS ISA:

- Jump to the effective target address in GPR *rs*. Bit 0 of GPR *rs* is interpreted as the target ISA Mode: if this bit is 0, signal an Address Error exception when the target instruction is fetched because this target ISA Mode is not supported. Otherwise, set bit 0 of the target address to zero, and fetch the instruction.

For processors that do implement the MIPS ISA:

- Jump to the effective target address in GPR *rs*. Set the ISA Mode bit to the value in GPR *rs* bit 0. Set bit 0 of the target address to zero. If the target ISA Mode bit is 0 and the target address is not 4-byte aligned, an Address Error exception will occur when the target instruction is fetched.

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.

JALRC.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 JALRC.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 JALRC.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.

Compact jumps do not have delay slots. The instruction after the jump is NOT executed when the jump is executed.

Availability and Compatibility:

Release 6 maps JR and JR.HB to JALRC and JALRC.HB with rt = 0:

Release 6 assemblers should accept the JR and JR.HB mnemonics, mapping them to the Release 6 instruction encodings.

Restrictions:

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 JALRC.HB, JALRSC.HB, JR.HB, ERET, or DERET. Further, the operation is **UNPREDICTABLE** if the mapping of the current instruction stream is modified.

Restrictions Related to Multiple Instruction Sets: This instruction can change the active instruction set, if more than one instruction set is implemented.

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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 MIPS and if the ISAMode bit of the target address is MIPS (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 MIPS ISA, if the intended target ISAMode is MIPS (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

Operation:

```
temp ← GPR[rs]
   GPR[rt] ← PC + 4
if Config3ISA = 1 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.

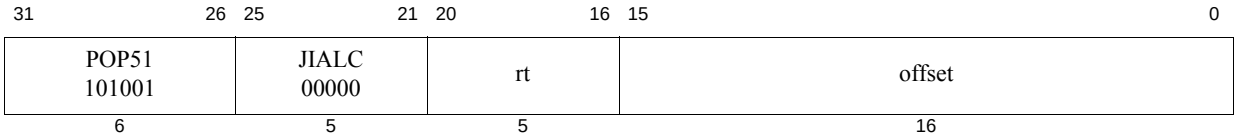
Release 6 `JR.HB rs` is implemented as `JALRC.HB r0,rs`. For example, as `JALRC.HB` with the destination set to the zero register, `r0`.

This instruction implements the final step in clearing execution and instruction hazards before execution continues. A hazard is created when a Coprocessor 0 or TLB write affects execution or the mapping of the instruction stream, or after a write to the instruction stream. When such a situation exists, software must explicitly indicate to hardware that the hazard should be cleared. Execution hazards alone can be cleared with the EHB instruction. Instruction hazards can only be cleared with a JR.HB, JALRC.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 */
   jalrc.hb a1                /* Call routine, clearing the hazard */
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `JIALC rt, offset` **microMIPS32 Release 6**

Purpose: Jump Indexed and Link, Compact

Description: `GPR[31] ← PC+4, PC ← ( GPR[rt] + sign_extend( offset ) )`

For processors that do not implement the MIPS ISA:

- Jump to the effective target address in GPR *rs*. Bit 0 of GPR *rs* is interpreted as the target ISA Mode: if this bit is 0, signal an Address Error exception when the target instruction is fetched because this target ISA Mode is not supported. Otherwise, set bit 0 of the target address to zero, and fetch the instruction.

For processors that do implement the MIPS ISA:

- Jump to the effective target address in GPR *rs*. Set the ISA Mode bit to the value in GPR *rs* bit 0. Set bit 0 of the target address to zero. If the target ISA Mode bit is 0 and the target address is not 4-byte aligned, an Address Error exception will occur when the target instruction is fetched.

The jump target is formed by sign extending the offset field of the instruction and adding it to the contents of GPR `rt`.

The offset is NOT shifted, that is, each bit of the offset is added to the corresponding bit of the GPR.

Places the return address link in GPR 31. The return link is the address of the following instruction, where execution continues after a procedure call returns. Compact jumps do not have delay slots. The instruction after the jump is NOT executed when the jump is executed.

Restrictions:

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR *rs*.

For processors that implement MIPS and if the ISAMode bit of the target is MIPS (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 MIPS ISA, if the intended target ISAMode is MIPS (bit 0 of GPR *rs* is zero), an Address Error exception occurs when the jump target is fetched as an instruction.

Availability and Compatibility:

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

Exceptions:

None

Operation:

```
temp ← GPR[rt] + sign_extend(offset)
   GPR[31] ← PC + 4
```

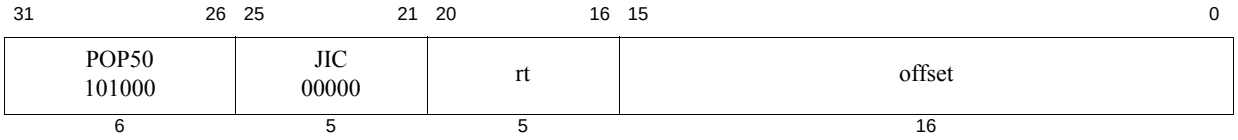

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
if Config3ISA = 1 then
       PC ← temp
    else
       PC ← (tempGPRLEN-1..1 || 0)
       ISAMode ← temp0
    endif
```

Programming Notes:

JIALC does NOT shift the offset before adding it the register. This can be used to eliminate tags in the least significant bits that would otherwise produce misalignment. It also allows JIALC to be used as a substitute for the JALX instruction, removed in Release 6, where the lower bits of the target PC, formed by the addition of GPR[rt] and the unshifted offset, specify the target ISAmode.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `JIC rt, offset` **microMIPS32 Release 6**

Purpose: Jump Indexed, Compact

Description: `PC ← ( GPR[rt] + sign_extend( offset ) )`

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.

The branch target is formed by sign extending the offset field of the instruction and adding it to the contents of GPR `rt`.

The offset is NOT shifted, that is, each bit of the offset is added to the corresponding of the GPR.

Compact jumps do not have a delay slot. The instruction after the jump is NOT executed when the jump is executed.

Restrictions:

Any instruction, including a branch or jump, may immediately follow a branch or jump, that is, Delay Slot restrictions do not apply in Release 6.

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.

Availability and Compatibility:

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

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

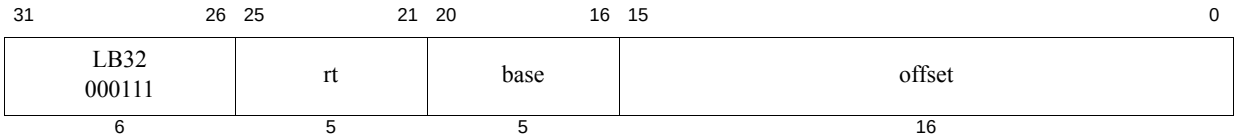
Operation:

```
temp ← GPR[rt] + sign_extend(offset)
if Config3ISA = 1 then
       PC ← temp
   else
       PC ← tempGPRLEN-1..1 || 0
       ISAMode ← temp0
   endif
```

Programming Notes:

JIC does NOT shift the offset before adding it the register. This can be used to eliminate tags in the least significant bits that would otherwise produce misalignment. It also allows JIALC to be used as a substitute for the JALX instruction, removed in Release 6, where the lower bits of the target PC, formed by the addition of GPR[rt] and the unshifted offset, specify the target ISAmode.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Load Byte

To load a byte from memory as a signed value.

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

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

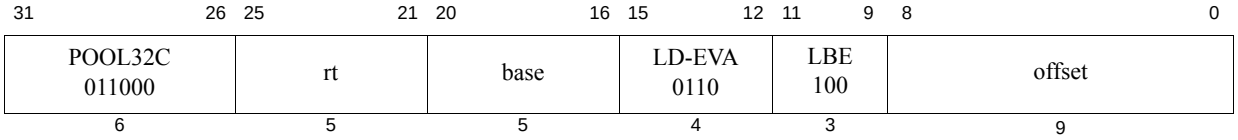
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1.. || (pAddr_..0 xor ReverseEndian)
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_..0 xor BigEndianCPU
GPR[rt] ← sign_extend(memword_7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← memory[GPR[base] + offset]`

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR *rt*. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

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

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

Restrictions:

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

Operation:

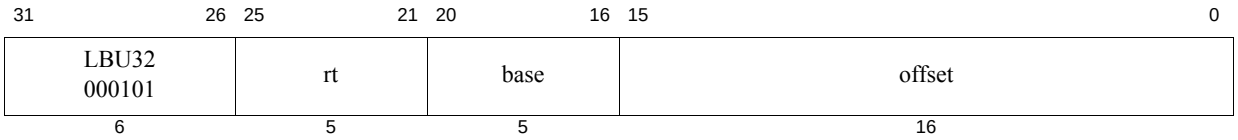
```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1.. || (pAddr_..0 xor ReverseEndian)
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_..0 xor BigEndianCPU
GPR[rt] ← sign_extend(memword_7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid

Bus Error, Address Error, Watch, Reserved Instruction, Coprocessor Unusable

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Load Byte Unsigned

To load a byte from memory as an unsigned value

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

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

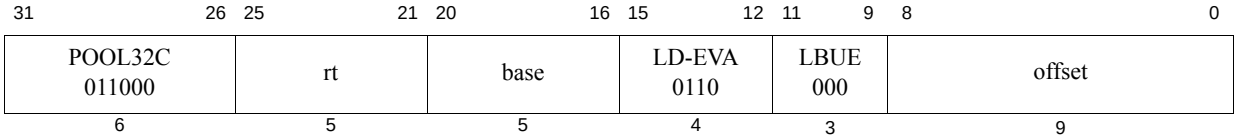
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1.. || (pAddr_..0 xor ReverseEndian)
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_..0 xor BigEndianCPU
GPR[rt] ← zero_extend(memword_7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

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

Restrictions:

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

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1.. || (pAddr_..0 xor ReverseEndian)
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_..0 xor BigEndianCPU
GPR[rt] ← zero_extend(memword_7+8*byte..8*byte)
```

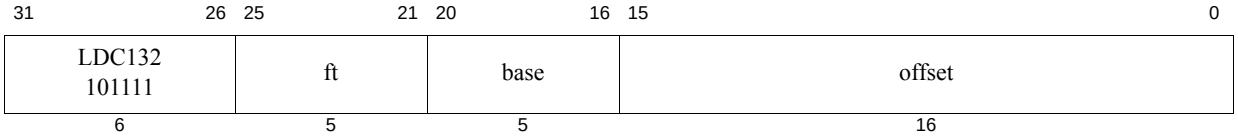
Exceptions:

TLB Refill, TLB Invalid

Bus Error, Address Error

Watch, Reserved Instruction, Coprocessor Unusable

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Load Doubleword to Floating Point

To load a doubleword from memory to an FPR.

Description: `FPR[ft] ← memory[GPR[base] + offset]`

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

Restrictions:

Pre-Release 6: An Address Error exception occurs if $EffectiveAddress_{2..0} \neq 0$ (not doubleword-aligned).

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

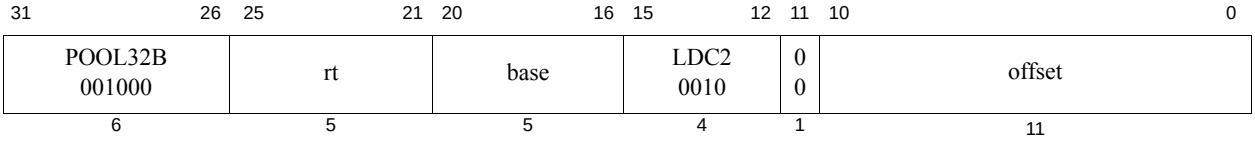
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
StoreFPR(ft, UNINTERPRETED_DOUBLEWORD, memdoubleword)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← memory[GPR[base] + offset]`

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in Coprocessor 2 register *rt*. The 9-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).

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
```

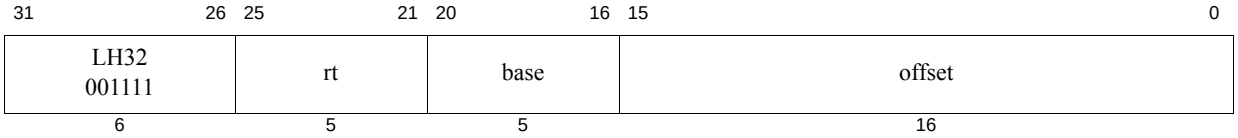
Exceptions:

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

Programming Notes:

Release 6 implements a 9-bit offset, whereas all release levels lower than Release 6 implement a 16-bit offset.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Load Halfword

To load a halfword from memory as a signed value

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

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

Restrictions:

Pre-Release 6: The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

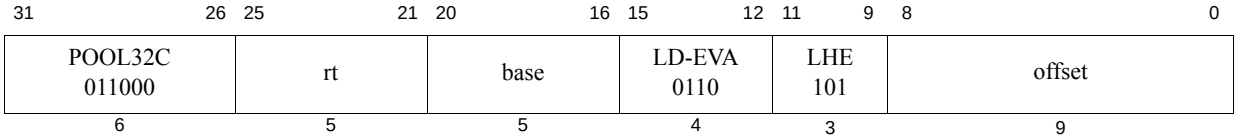
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1.. || (pAddr_..0 xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr_..0 xor (BigEndianCPU || 0)
GPR[rt] ← sign_extend(memword_15+8*byte..8*byte)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← memory[GPR[base] + offset]`

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, and placed in GPR *rt*. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The LHE instruction functions the same 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.

Pre-Release 6: The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

Operation:

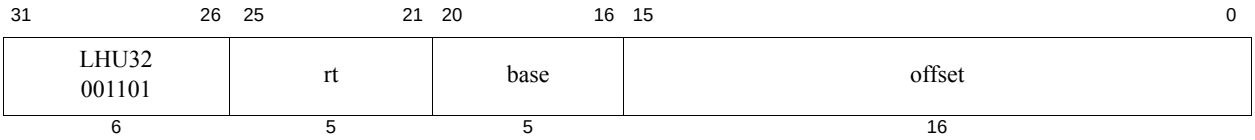
```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1.. || (pAddr_..0 xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr_..0 xor (BigEndianCPU || 0)
GPR[rt] ← sign_extend(memword_15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Watch, Reserved Instruction, Coprocessor Unusable

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Load Halfword Unsigned

To load a halfword from memory as an unsigned value

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

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

Restrictions:

Pre-Release 6: The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

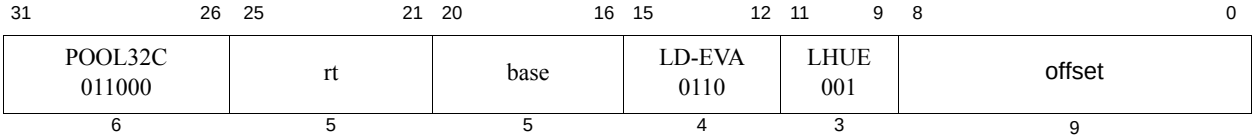
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]

endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1.. || (pAddr..0 xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr..0 xor (BigEndianCPU || 0)
GPR[rt] ← zero_extend(memword_15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

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

Restrictions:

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

Pre-Release 6: The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

Operation:

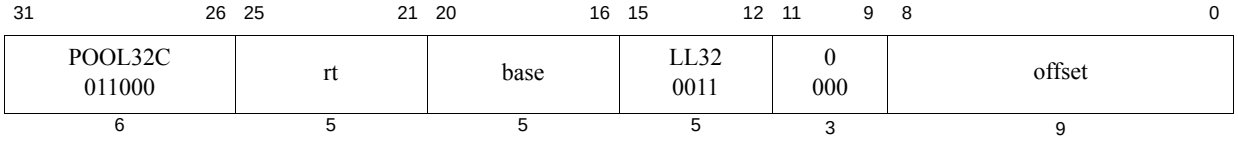
```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1.. || (pAddr..0 xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr..0 xor (BigEndianCPU || 0)
GPR[rt] ← zero_extend(memword_15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Watch, Reserved Instruction, Coprocessor Unusable

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← memory[GPR[base] + offset]`

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

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and written into GPR *rt*. The 9-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 nonzero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (AddressError) on a misaligned access is required.

Operation:

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

Exceptions:

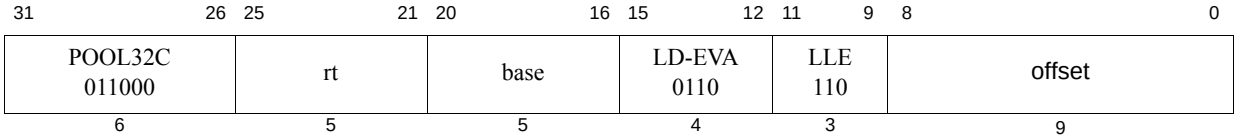
TLB Refill, TLB Invalid, Address Error, Watch

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Programming Notes:

Release 6 implements a 9-bit offset, whereas all release levels lower than Release 6 implement a 16-bit offset.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← memory[GPR[base] + offset]`

The LLE and SCE instructions provide the primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations using user mode virtual addresses while executing in kernel mode.

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and written into GPR *rt*. The 9-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 the same 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.

Pre-Release 6: The effective address must be naturally-aligned. If either of the 2 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (AddressError) on a misaligned access is required.

Operation:

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


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

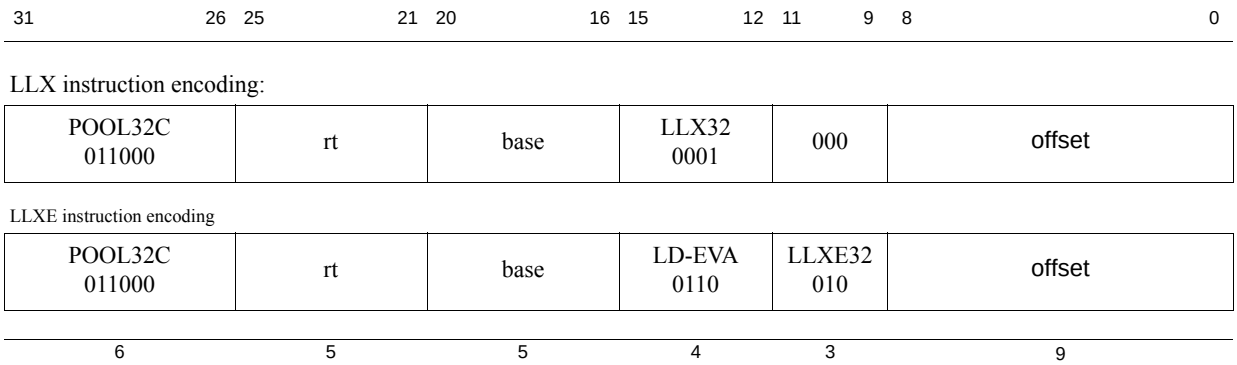
```
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
←GPR[rt] ← memword
LLbit ← 1
```

Exceptions:

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

Programming Notes:

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `LLX, LLXE`

`LLX rt, offset(base)` **microMIPS32 Release 6**

`LLXE rt, offset(base)` **microMIPS32 Release 6**

Purpose: Load Linked Extended {Word,Word EVA}

Load from memory, extending following Load Linked; word, or word EVA

Description:

The LLX/SCX family of instructions (LLX, LLXE, SCX, SCXE) extends the MIPS LL/SC mechanism for performing atomic read-modify-writes to permit more than one memory location to be accessed atomically. The memory locations are constrained to be aligned, adjacent and within both the same synchronization block and the same cache line (if applicable).

LL-SC and LLE-SCE allow 32-bit aligned atomic memory operations to be performed on MIPS32. LLX/LL-SCX/SC and LLXE/LLE-SCXE/SCE allow 64-bit aligned atomic memory operations to be performed on MIPS32.

LL-SC code sequences in general, and LLX/LL-SCX/SC in particular, provide atomicity if the computer system can guarantee that, if the SC succeeds, then atomicity has not been violated by operations between the LL and SC. It should also guarantee eventual success, i.e. that failures will not persist forever.

An LLX family instruction (LLX/LLXE) (at PC) must be followed by a matching LL family instruction (LL/LLE) (at PC+4), forming an LLX/LL instruction family pair (LLX/LL, LLXE/LLE). See **Restrictions** section for a full description of match requirements, and special case for SDBBP and BREAK breakpoint instructions.

The signed *offset* is added to the contents of GPR *base* to form an effective address. This address must be naturally aligned.

The memory bytes accessed by the LLX family instruction and the following, matching LL family instruction must be adjacent, non-overlapping, and aligned. The following, matching, LL family instruction must be aligned to double the access width. I.e. in an LLX/LL pair, the LL instruction must be aligned to an 8-byte boundary, and the LLX data address must be 4 bytes higher; similarly for an LLXE/LLE pair, the LLE instruction must be aligned to an 8-byte boundary, and the LLXE data address must be 4 bytes higher.

For LLX and LLXE: the 32-bit word at the memory location specified by the effective address is fetched, and written into GPR *rt*.

If the LLX family instruction is followed by a matching LL family instruction, behavior is as if a double width load access suitable for starting an atomic sequence is performed[1]. Memory data corresponding to the low byte addresses returned is written to GPR *rt* of the LL family instruction; the part corresponding to high byte addresses is written to GPR *rt* of the LLX instruction.

1. It is implementation dependent whether a single double width access, or two separate normal width accesses, are performed.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

An LLX/LL family instruction pair (LLX/LL, LLXE/LLE) begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor. Any subsequent LL family instruction or LLX/LL family instruction pair, when executed, starts an active RMW sequence replacing any other sequence that was active. The RMW sequence for an LLX/LL family instruction pair is completed by a subsequent SCX/SC family instruction pair, which should match the LLX/LL pair in type and size, and which either completes the RMW sequence atomically and succeeds, or does not and fails.

If the PC and PC+4 instruction encodings do not match, a Reserved Instruction exception is signaled. If the effective addresses of the LLX/LL or LLXE/LLE family instruction pair are not 32-bit word aligned separately and 64-bit doubleword aligned together, then Address Error is signaled. If the effective address of the following LL family instruction (at PC+4) is not the lowest byte address, then an Address Error exception is signaled. See **Restrictions** section for a full description of match requirements, and special case for SDBBP and BREAK breakpoint instructions.

If an exception occurs between the LLX family instruction at PC and the instruction at PC+4 (LL family, SDBBP or BREAK, or non-matching instruction which will signal a Reserved Instruction exception), the exception is reported with EPC=PC and Status.BD=1. In this case the LLX family instruction will have partially executed: exceptions relating solely to the LLX family instruction in isolation will already have been reported, including Address Error and TLB exceptions, but the actual memory reference will not yet have been performed, since it can only be performed atomically in conjunction with the following LL family instruction. The target register of the LLX family instruction will NOT have been updated. However, LLbit will be clear on entry to the exception handler, even if LLbit was set before the LLX family instruction started.[2]

Executing an LLX/LL family instruction pair on one processor does not cause an action that, by itself, causes an SC or SCX/SC pair for the same block to fail on another processor.

An execution of an LLX/LL family instruction pair does not have to be followed by execution of a matching SCX/SC instruction pair; a program is free to abandon the RMW sequence without attempting a write.

Restrictions:

The following restrictions apply to load-linked and store-conditional extended instructions in the LLX/SCX instruction family:

Coprocessor 0's *Cause* register bit *BD* is extended to indicate exceptions related to the next instruction after the LLX/SCX-family instruction. Pseudocode indicates what value *Cause.BD* should be set to via comments such as `SignalException(AddressError) /*BD=1*/`. Similarly, the status register *BadInstrP* is extended to hold the LLX/SCX-family instruction if an exception is signaled for the next instruction, with *BD*=1.

An LLX/SCX family instruction must be not be placed in a branch delay slot or compact branch forbidden slot: if this rule is violated, a Reserved Instruction exception will be signaled (with *EPC*=PC of branch, *BD*=1).

An LLX/SCX family instruction must be followed by a matching LL/SC-family instruction: An SCX instruction must be followed by an SC instruction of the same type. Similarly for LLX/LL, LLXE/LLE, and SCXE/SCE. If the following instruction does not match, a Reserved Instruction exception must be signaled (with *EPC*=PC of the LLX/SCX family instruction, *BD*=1).

Except: An LLX/SCX instruction may be followed by one of the breakpoint instructions BREAK or SDBBP, in which case the appropriate breakpoint exception takes priority over the Reserved Instruction exception. The BREAK exception will be signaled with *EPC*=PC of the LLX/SCX family instruction and *BD*=1. The debug exception caused by such an SDBBP will be reported with *DEPC*=PC of the LLX/SCX family instruction and *DBD*=1.

The *base* field must be the same in an LLX/SCX family instruction and the following, matching, LL/SC-family instruction: If the following instruction does not match, a Reserved Instruction exception must be signaled (with *EPC*=PC of the LLX/SCX family instruction, *BD*=1).

2. E.g. LLX rt, mem; Trap... SC => LLX's rt is not updated, but the SC is required to fail unless the trap handler has successfully completed the LLX/LL family instruction pair.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

The *base* and *rt* fields of the LLX family instruction must not be the same. If they are the same a Reserved Instruction exception must be signaled (with *EPC*=PC of the LLX/SCX family instruction, *BD*=0).

The LLX/SCX and following LL/SC family instructions must match in their *offset* field: Given matching in instruction type and *base*, the difference between the *offset* fields of the instruction at PC and the instruction at PC+4 should be the data size, 4 for LLX/LLE/SCX/SCXE. Programmers should follow this rule in coding. However, implementations do not need to explicitly check this rule, since it is implied by other rules. TBD

Natural Alignment: The effective address must be naturally aligned for any LLX/SCX family instruction; if not naturally aligned, an Address Error exception is signaled. I.e. for LLX, LLXE, SCX and SCXE, if the two least significant bits of the effective address are not both zero, an Address Error exception is signaled. Such an Address Error exception is signaled with *EPC*=PC of the LLX/SCX family instruction, BD=0.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (AddressError) on a misaligned access is required.

Double Width Alignment: In addition to natural alignment, the memory bytes written by the LLX/SCX family instruction and the following LL/SC family instruction must be adjacent, non-overlapping, and must have the alignment natural for double the memory access size: The lowest byte address in an LLX/LL, LLXE/LLE, SCX/SC or SCXE/SCE pair must be 8-byte aligned. It is required that the LL/SC family instruction byte address be lower than that of the LLX/SCX family instruction. i.e. that the LL/SC family instruction in an LLX/LL or SCX/SC family instruction pair must be naturally aligned for double the memory access width.

The double width alignment condition must be satisfied for both virtual and physical addresses. If this condition is not met, then an Address Error exception is signaled, with EPC = PC of first instruction, and BD=1. This condition is guaranteed to be met in the physical address if met in the virtual address and if the SCX and SC translations are consistent.

Exception Priority: although LLX and LL may complete execution together, all exceptions for an LLX instruction (at PC) must be signaled, with *EPC*=PC and *BD*=0, before any exceptions are signaled, with *EPC*=PC and *BD*=1, for the next instruction (at PC+4) or for any exceptions caused by the interaction between the LLX instruction and the next instruction. This is as if the LLX instruction is executed enough to signal all exceptions, followed by exception checks for the combination of LLX and the next instruction. Similarly for LLX/LL, LLXE/LLE, and SCXE/SCE instructions.

Exceptions relating to an LLX/SCX family instruction are reported with *EPC*=PC of the LLX/SCX family instruction, and *BD*=0.

Exceptions relating to interaction between an LLX/SCX family instruction and the following instruction are reported with *EPC*=PC of LLX/SCX instruction and *BD*=1.

Debug single step exceptions are reported with *DEPC*=PC of the LLX/SCX family instruction, and *BD*=0. No debug single step exception will be reported for the SC instruction of an SCX/SC pair: For the purposes of debug single stepping, the SCX/SC pair is atomic. Similarly for LLX/LL, LLE/LLXE, and SCXE/SCE pairs of instructions.

Exceptions related to the SCX/SC family instruction pair before following instruction cancel SCX but do ***not*** clear *LLbit*: if an exception or interrupt occurs at or after the SCX-family instruction and before or at the next instruction, the SCX is canceled, but *LLbit* is not cleared. I.e. the LLX/LL-SCX/SC atomic is not necessarily forced to fail. Exceptions are therefore reported with *EPC*=PC of SCX, and *BD*=0 or 1 as appropriate. Exception handling software should return (ERET or ERETNC) to the PC of the SCX instruction, re-executing the SCX/SC pair. Adjusting EPC or DEPC and returning to the SC instruction without re-executing the SCX instruction will result in incorrect behavior.

For exceptions related to an LLX/LL family instruction pair:

- No memory access is performed.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

- Neither target register of the LLX/LL family instruction pair is updated.
- *LLbit* is not set.
- *EPC* (or *DEPC*) is set to the PC of the LLX family instruction.
- Status.BD is set to 0 or 1 as appropriate, as described below.

Exception handling software should return (ERET or ERETNC) to the PC of the LLX instruction, re-executing the LLX/LL pair. Adjusting EPC or DEPC and returning to the LL instruction without re-executing the LLX instruction will result in incorrect behavior.

LLX/LL and SCX/SC matching: the LL-family instruction, the SC-family instruction, and the optional LLX/SCX-family instructions in a MIPS atomic sequence *should*[3] match. Portable software should not rely on mismatching LLX/LL/SCX/SC to complete successfully, nor to fail. Implementations are permitted to cause the SC to fail if the LL/SCX/SC do not match, but are not required to do so. Matching LLX/LL/SCX/SC should be of the same instruction type (word (LLX/LL/SCX/SC), or word EVA (LLXE/LLE/SCXE/SCE)). Table 5.25 summarizes these rules for LL/SC family instructions.

Table 5.25 Recommended and non-recommended LL/SC family instructions to start and end atomic code sequences

		Start of atomic sequence					
		LL	LLD	LLE	LLX /LL	LLDX /LLD	LLXE /LLE
End of Atomic Sequence	SC	**OK**[1]	BAD	BAD	BAD	BAD	BAD
	SCD	BAD[2]	**OK**	BAD	BAD	BAD	BAD
	SCE	BAD	BAD	**OK**	BAD	BAD	BAD
	SCX/SC	BAD	BAD	BAD	**OK**	BAD	BAD
	SCDX/SCD	BAD	BAD	BAD	BAD	**OK**	BAD
	SCXE/SCE	BAD	BAD	BAD	BAD	BAD	**OK**

1. Cells marked OK indicate recommended combinations of instructions to start and end LL/SC atomic code sequences.
2. Cells marked BAD (and shaded) indicate non-recommended combinations of instructions to start and end LL/SC atomic code sequences. Software should not be coded in this way. Implementations are not required to enforce this restriction, but software coded this way may succeed on some implementations, and fail on other implementations. I.e. success or failure of the SC family instruction is UNPREDICTABLE.

The LL and SC virtual and physical addresses should match completely. However, the memory addressing mode - the and offset - need not match between LLX/LL and SCX/SC. All physical address bits in the LL physical address and the corresponding bits in the SC physical address should match to the alignment required for the size of the LL/SC family instructions or LLX/LL and SCX/SC family instruction pairs.[4] This applies to atomic code sequences created

3. Terminology: "*Should*" is a recommendation. Implementations are encouraged to provide *should* behavior, but are not required to do so. Portable software should not rely on such behavior, but is encouraged to follow *should* rules. "*Must*" behavior are requirements: Implementations are required to implement such behavior, and software that violates such requirements will fail, typically with a exception such as a Reserved Instruction exception or Address Error.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

via LL/SC, LLE/SCE, and their corresponding extended versions LLX/LL-SCX/SC, LLXE/LLE-SCXE/SC.

Translation Consistency: It is required that LL and SC match addresses, and that LLX/SCX family instructions lie in the same synchronization block. Even if all virtual addresses match, on a processor with hardware page table walking it is possible for physical address translation to change between LL and SC, and between the execution phase of LLX, LL, SCX and SC family instructions. e.g. between the time that SCX is first executed, and the time that the SCX store data is committed along with SC. The SCX/SC must only succeed if the SCX and SC physical addresses are consistent. If the address translations are inconsistent, implementations are required to fail the SCX/SC pair, or to retry them in a manner transparent to software. Similarly for LLX/LL pairs. Similarly for other information obtained from translation, such as the CCA (Cacheability and Coherence Attribute).

It is required that LLX/LL or SCX/SC instruction pairs act as if only a single address translation is done for the first instruction in the pair, and that translation is used for the second instruction, changing only lower address bits 3:0. Similarly for LLX/LL, LLXE/LLE, and SCXE/SCE instruction pairs.

Synchronizable memory type (CCA): 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.

LLX/LL need not be writeable: The addressed location need not be writable for LL or LLX family instructions. If it is not writable a subsequent SC or SCX family instruction will fault, but LL or LLX family instructions may be used in situations that do not generate such faults, e.g. the PAUSE instruction.

LLX/LL and PAUSE: If an LLX/LL family instruction pair is followed by a PAUSE instruction, the PAUSE instruction must terminate if it cannot be guaranteed that any of the memory bytes address by the LLX/LL instruction pair have not been modified.

Memory Ordering of LL/SC family instructions (included LLX/SCX family instructions):

- An SCX/SC family instruction pair is executed atomically as seen by the processor executing these instructions and by other processors. I.e. the SC will not be seen to be executed before the SCX, and no other instruction, processor or device, can observe the SCX store without also being able to observe the SC store, or vice versa.
- LLX/LL family instruction pairs are not required to perform a double width atomic read of memory, but violations of atomicity will be detected, clearing LLbit, so that the matching SC will fail.[5]
    - Atomicity of LLX/LL family instruction pairs may be provided by MIPS CPU implementations as and if required by certain system configurations for uncached memory. [6]

4. Note that the implementation dependent *LLAddr* register (Load Linked Address (CP0 Register 17, Select 0)) does not hold physical address bits 0 to 4 as of Release 5 or after. The requirement all LL and SC address bits match therefore involves comparing LL address bits not stored in any software accessible register state.
5. For example, an implementation of LLX/LL in cached memory may have LLX set LLaddr and then perform the LLX word load, and then may execute LL separately. A separate processor may perform an atomic doubleword write that changes both the LLX and LL memory locations, such that the values returned by LLX and LL may not have both been simultaneously present in memory. However, if atomicity is violated in this way, then LLbit must be cleared. The LL instruction of an LLX/LL instruction pair will not set LLbit if it has been cleared after the LLX instruction. Overall, LLX/LL family instruction pairs are not required to be atomic; whereas SCX/SC family instruction pairs are required to be atomic, if performed.
   However, certain system configurations, for uncached memory in particular, require that the LLX/LL family instruction pair be performed atomically via a single bus transaction.
6. MIPS recommends that implementations perform a double width atomic read memory access for LLX/LL family instruction pairs, for cached as well as uncached memory, but does not require this. Portable software should not assume that an LLX/LL family instruction pair is atomic without using a matching SCX/SC family instruction pair to detect possible violations of atomicity.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

- All LL/SC family instructions, including LLX/LL and SCX/SC family instruction pairs, are ordered by their implicit dependency on LLbit: e.g. a later LL will not be executed before an earlier SC from the same processor, even if their data memory addresses do not overlap.

- In the MIPS memory consistency architecture, LL/SC family instructions (including LLX/SCX family instructions) are not ordered with respect to other memory accesses from the same processor, except when their addresses overlap, or explicit SYNC instructions lie between them. E.g. a later LL can be executed before an earlier SW, or vice versa.[7]

An LLX family instruction should not overwrite its own base register: code sequences such as that below

```
LLX r10, (r10)4
LL  r8,  (r10)0
```

where the *rt* and *base* fields of an LLX family instruction specify the same GPR are discouraged.

LLX/LL family instruction pair writing the same target GPR *rt*: in code sequences such as that below

```
LLX r4, (r10)4
LL  r4, (r10)0
```

where the *rt* fields are the same for both members of an LLX/LL family instruction pair, the value loaded and written by the last instruction, the LL family member, will be the value written. The value loaded and supposedly written into the register by the first instruction, the LLX family member, is not directly observable: if an exception prevents the LL from executing, the LLX target register is not written.

Availability and Compatibility:

The LLX/SCX instruction family is introduced by and required as of the MIPS Release 6 and microMIPS Release 6 architecture.

LLX and SCX are introduced by and required as of microMIPS32 Release 6. LLXE and SCXE are introduced by and required as of microMIPS32 Release 6 when EVA is also implemented, which is indicated by bit *EVA* of coprocessor 0's *Config5* register.

The microMIPS Release 6 instruction encodings for the LLX family of instructions conflict with encodings used by valid instructions in microMIPS pre-Release 6: LLX conflicts with pre-Release 6 LWR, and LLXE conflicts with pre-Release 6 LWLE.

7. Note that this applies also to ordinary load instructions lying between LL and SC, inside the atomic RMW sequence.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Operation:

```
/* pseudocode for LLX and for the following instruction;
 * this replaces the following instruction pseudocode.
 *
 * this_instruction = LLX instruction at PC during instruction time I
 * next_instruction = instruction at PC+4 during instruction time I
 *                  = instruction at PC during instruction time I+1
 *                  = LL, or BREAK or SDBBP, else invalid
 * 'LLX' and 'LL' are generic, applicable to LLX-family and LL-family.
 *
 * All exceptions are signaled with EPC or DEPC = PC of LLX instruction.
 * All exceptions in instruction time I are signaled with BD=0.
 * All exceptions in instruction time I+1 are signaled with BD=1.
 */
I: /* LLX-only execution in instruction time I */
   /* perform address calculation and translation and LLX-only checks. */

   /* LLbit is set only on successful completion;
    * LLbit is cleared after all unsuccessful completions of LLX/LL pairs
    * including when exceptions are signalled
    * (unlike all other situations, where exceptions do not affect LLbit)
    */

   if this_instruction is LLX then
      size ← 4
   else if this_instruction is LLXE then
      EVA_Checks() /*BD=0*/
      size ← 4
   else
      assert(IMPOSSIBLE)
   endif

   /* LLX family instructions must not write their base register */
   if this_instruction.base ≠ this_instruction.rt
      then SignalException(ReservedInstruction) /*BD=0*/ endif

   this_va ← GPR[this_instruction.base] + sign_extend( this_instruction.offset )
   if this_va & (size-1) ≠ 0 then SignalException(AddressError) /*BD=0*/ endif

   /* AddressTranslation of first instruction
    * will be used for the second instruction as well,
    * changing lower address bits,
    * to avoid translation consistency issues */
   (this_pa,this_cca) ← AddressTranslation( this_va, DATA, LOAD) /*BD=0*/

   /* complete LLX execution in instruction time I+1 */


I+1:
   /* LLX execution time I+1 and next_instruction execution time I combined */
   /* All exceptions in instruction time I+1 are signaled with BD=1. */

   LLX_SCX_family_common_code(
          /*in:*/ this_instruction, this_pa, this_cca, size,
          /*out:*/ next_instruction, next_va, next_pa, next_cca
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
    )

    /* Actual execution of the double-width LLX/LL family instruction pair
     * LLX/LL // LLXE/LLE */
    /* note that next_pa is derived from this_pa[8] */
    memdoubleword ← LoadMemory(next_cca, 8, next_pa, next_va, DATA)
                        /* extended for special uncached bus transaction */
    if BigEndianCPU then
       GPR[this.rt] ← memdoubleword31..0
       GPR[next.rt] ← memdoubleword63..32
    else
       GPR[this.rt] ← memdoubleword63..32
       GPR[next.rt] ← memdoubleword31..0
    endif /* endianness */

    /* LLbit is set only on successful completion;
     * LLbit is cleared after all unsuccessful completions of LLX/LL pairs
     * including when exceptions are signalled
     * (unlike all other situations, where exceptions do not affect LLbit)
     */
    LLbit ← 1

/* end of combined LLX/ LLpseudocode */

where /* helper pseudocode */

function EVA_checks(vaddress)
    if (Config5EVA=0) then SignalException(ReservedInstruction) endif
    if !IsCoprocessorEnabled(0)
       then SignalException(CoprocessorUnusable, 0)endif
    AM = SegmentAM(vaddress)
    if (AM != UUSK && AM != MUSK && AM != MUSUK)
       then SignalException(AddressError) endif
end function
```

8. Note that LLX_SCX_common_code() sets next_pa = this_pa-size = this_pa & (size-1), assuming all other constraints are met. Only a single address translation is required.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
function LLX_SCX_family_common_code (
    /*inputs: */ this_instruction, this_pa, this_cca, size,
    /*outputs:*/ next_instruction, next_va, next_pa, next_cca
)
    /* begin function */
    if next_instruction is BREAK or SDBBP then
        /* Execute BREAK or SDBBP in normal I+1 manner,
         * as if in a branch delay slot or compact branch forbidden slot.
         * signaling appropriate exception */
    endif

    /* next_instruction must be matching non-extended LL/SC family
     * - this pseudocode replaces normal pseudocode for next instruction. */
    if (this_instruction is LLX and next_instruction is not LL)
        or (this_instruction is LLXE and next_instruction is not LLE)
        or (this_instruction is SCX and next_instruction is not SC)
        or (this_instruction is SCXE and next_instruction is not SCE)
    then
        SignalException(ReservedInstruction) /*BD=1*/
endif
    /* next instruction is non-extended LL/SC family: consistency checks */

    /* Check base register field for consistency */
    if this_instruction.base ≠ next_instruction.base
        then SignalException(ReservedInstruction) /*BD=1*/ endif

    /* Address computation for LL/SC-family next_instruction */
    next_va ← GPR[next_instruction.base] + sign_extend( next_instruction.offset )

    /* LL/SC following LLX/SCX virtual address must be doublewidth aligned
    if next_va & (size*2-1) ≠ 0
        then SignalException(AddressError) /*BD=1*/ endif

    /* LLX/SCX and LL/SC address virtual addresses must be adjacent
     * (adjacent, nonoverlapping, doubleword aligned) */
    if this_va&(2*size-1) - next_va&(2*size-1) ≠size
        then SignalException(AddressError) /*BD=1*/ endif
    /* assert( this_va-next_va ≠ size ) */

    /* Check offsets for consistency */
    /* assert( this_instruction.offset - next_instruction.offset = size ) */
        /* offset check not needed - other constraints ensure */

    /* LL/SC virtual to physical address translation
    /* Reuse the translation of the first instruction to ensure consistency. */
    /* Note: after all RI and AE exceptions, for standard exception priority. */
    next_pa ← this_pa & (2*size-1)
        /* given alignment constraints,
         * next_pa = this_pa - size = this_pa & (2*size-1) */
    next_cca ← this_cca

end function /* LLX_SCX_family_common_code */
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

Reserved Instruction

Programming Notes:

Implementation Notes:

The synchronization block of memory used for LL/SC (and when extended by LLX/SCX) is typically the largest cache line in use.

Implementations of LL/SC in general, and LLX/LL-SCX/SC in particular, provide atomicity if the computer system can guarantee that, if the SC passes, then atomicity has not been violated by transactions between the LL and SC. It should also guarantee eventual success, i.e. that failures will not persist forever.

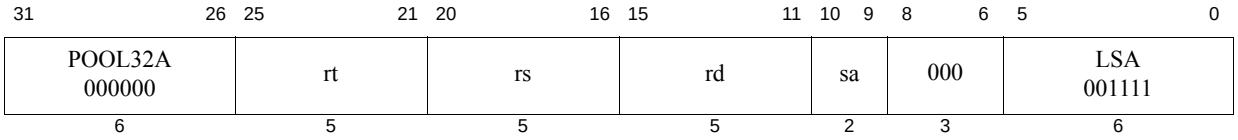
Correct implementation depends on the system, both the CPU and the external memory subsystem. For example, the CPU may implement LL/SC correctly for cacheable coherent memory, but if the I/O subsystem can write to memory without being exposed to the cache coherency mechanism, LL/SC will not detect violations of atomicity caused by such non-coherent I/O accesses. Similarly, the CPU may implement uncached memory requests for LL and SC, but if the external memory subsystem performs an SC request and returns success without guaranteeing atomicity, LL/SC may not provide the expected guarantee of atomicity.

If it is not possible to guarantee such atomicity then it is recommended that implementations cause the SC to fail, returning the failure code in GPR[rt] without performing the store.

LL/SC and LLX/LL-SCX/SC code sequences should only be used for the following memory types (Cache and Coherency Attributes (CCAs)):

- *cached coherent*: if the cache protocol can guarantee that atomicity has not been violated by transactions between the LL and SC.
- *uncached:*
  - for uncached memory that is memory-like, i.e. which does not have memory-mapped I/O side effects
  - if the CPU supports bus transactions visible to external hardware so that such external hardware can guarantee that atomicity has not been violated by transactions between the LL and SC, and can signal success or failure by replying to the uncached bus transaction triggered by the SC-family instruction.
  - or if the system configuration is such that the CPU can observe all memory transactions that would violate atomicity
- *cached noncoherent* or *uncached* (no side effects): on uniprocessor systems lacking cache coherence or external hardware that can make atomicity assertions, LL-SC and LLX/LL-SCX/SC code sequences can be used to detect violations of atomicity caused by interrupt handling
- for other memory types: it may be **UNPREDICTABLE** whether the SC and possible SCX stores are performed, and whether the SC reports success or failure.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `LSA`
`LSA rt, rs, rd, sa` **microMIPS32 Release 6**

Purpose: Load Scaled Address

Description:

```
LSA:   GPR[rd] ← sign_extend.32( (GPR[rs] << (sa+1)) + GPR[rt] )
```

LSA adds two values derived from registers `rs` and `rt`, with an optional scaling shift on `rs`. The scaling shift is formed by adding 1 to the 2-bit sa field, which is interpreted as unsigned. The scaling left shift varies from 1 to 5, corresponding to multiplicative scaling values of ×2, ×4, ×8, ×16, bytes, or 16, 32, 64, or 128 bits.

LSA is a MIPS32 compatible instruction, sign extending its result from bit 31 to bit 63.

Restrictions:

None

Availability and Compatibility:

LSA instruction is introduced by and required as of Release 6.

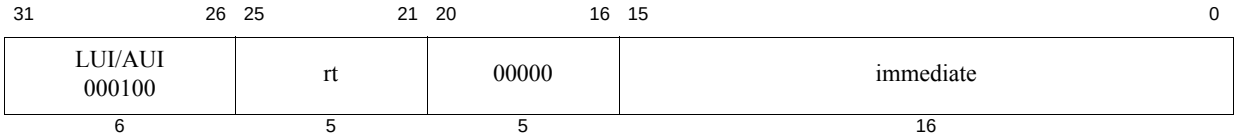
Operation

```
LSA:   GPR[rd] ← sign_extend.32( GPR[rs] << (sa+1) + GPR[rt] )
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `LUI rs, immediate` **microMIPS, Assembly Idiom Release 6**

Purpose: Load Upper Immediate

To load a constant into the upper half of a word

Description: `GPR[rs] ← immediate || 0^16`

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

Restrictions:

None

Operation:

```
GPR[rs] ← immediate || 0^16
```

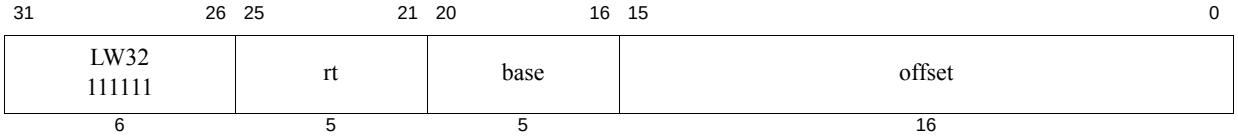
Exceptions:

None

Programming Notes:

In Release 6, LUI is an assembly idiom of AUI with rs=0.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Load Word

To load a word from memory as a signed value

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

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

Restrictions:

Pre-Release 6: The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

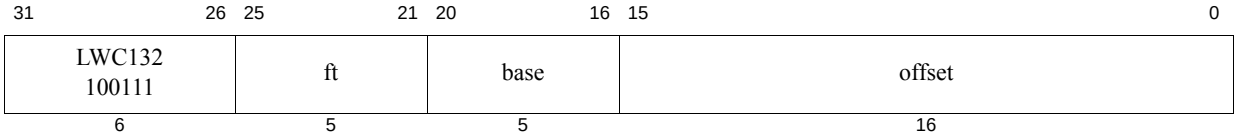
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt]  memword
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Load Word to Floating Point

To load a word from memory to an FPR

Description: `FPR[ft] ← memory[GPR[base] + offset]`

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of FPR *ft*. If FPRs are 64 bits wide, bits 63..32 of FPR *ft* become **UNPREDICTABLE**. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

Pre-Release 6: An Address Error exception occurs if $\text{EffectiveAddress}_{1..0} \neq 0$ (not word-aligned).

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

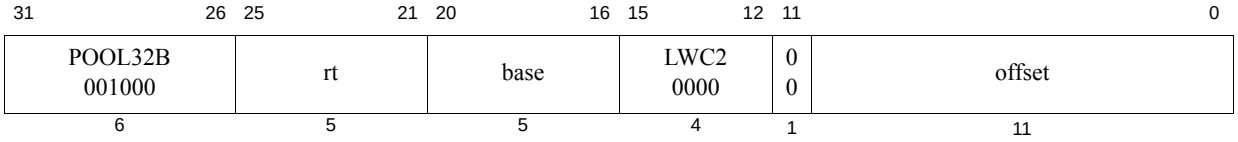
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)

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

StoreFPR(ft, UNINTERPRETED_WORD,
          memword)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Load Word to Coprocessor 2

To load a word from memory to a COP2 register.

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

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

Restrictions:

Pre-Release 6: An Address Error exception occurs if +EffectiveAddress$_{1..0} \neq 0$ (not word-aligned).

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

Operation:

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

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

CPR[2,rt,0] ← memword
```

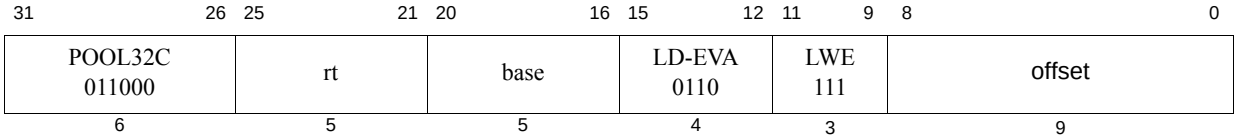
Exceptions:

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

Programming Notes:

Release 6 implements an 11-bit offset, whereas all release levels lower than Release 6 implement a 16-bit offset.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← 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 the same 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.

Pre-Release 6: The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

Operation:

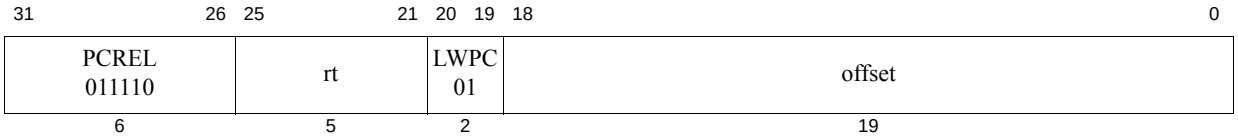
```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Watch, Reserved Instruction, Coprocessor Unusable

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `LWPC rt, offset` **microMIPS32 Release 6**

Purpose: Load Word PC-relative

To load a word from memory as a signed value, using a PC-relative address.

Description: `GPR[rt] ← memory[ PC & ~0x3 + sign_extend( offset << 2 ) ]`

The offset is shifted left by 2 bits, sign-extended, and added to the address of the LWPC instruction.

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

Restrictions:

None.

LWPC is naturally aligned, by specification.

Availability and Compatibility:

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

Operation

```
vAddr ← ( PC & ~0x3 + sign_extend(offset)<<2)
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
```

Exceptions:

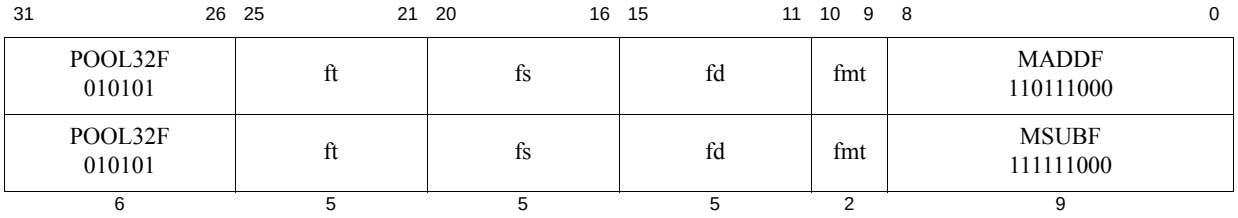
TLB Refill, TLB Invalid, TLB Read Inhibit, Bus Error, Address Error, Watch

Programming Note

The Release 6 PC-relative loads (LWPC) are considered data references.

For the purposes of watchpoints (provided by the CP0 *WatchHi* and *WatchLo* registers) and EJTAG breakpoints, the PC-relative reference is considered to be a data reference rather than an instruction reference. That is, the watchpoint or breakpoint is triggered only if enabled for data references.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



31–26	25–21	20–16	15–11	10–9	8–0
POOL32F 010101	ft	fs	fd	fmt	MADDF 110111000
POOL32F 010101	ft	fs	fd	fmt	MSUBF 111111000
6	5	5	5	2	9

Format: `MADDF.fmt MSUBF.fmt`

`MADDF.S fd, fs, ft` **microMIPS32 Release 6**

`MADDF.D fd, fs, ft` **microMIPS32 Release 6**

`MSUBF.S fd, fs, ft` **microMIPS32 Release 6**

`MSUBF.D fd, fs, ft` **microMIPS32 Release 6**

Purpose: Floating Point Fused Multiply Add, Floating Point Fused Multiply Subtract

MADDF.fmt: To perform a fused multiply-add of FP values.

MSUBF.fmt: To perform a fused multiply-subtract of FP values.

Description:

```
MADDF.fmt: FPR[fd] ← FPR[fd] + (FPR[fs] × FPR[ft])
MSUBF.fmt: FPR[fd] ← FPR[fd] - (FPR[fs] × FPR[ft])
```

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

(For MSUBF.fmt, the product is subtracted from the value in FPR *fd*.)

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

Restrictions:

None

Availability and Compatibility:

MADDF.fmt and MSUBF.fmt are required in Release 6.

MADDF.fmt and MSUBF.fmt are not available in architectures pre-Release 6.

The fused multiply add instructions, MADDF.fmt and MSUBF.fmt, replace pre-Release 6 instructions such as MADD.fmt, MSUB.fmt, NMADD.fmt, and NMSUB.fmt. The replaced instructions were unfused multiply-add, with an intermediate rounding, although in some earlier implementations they were fused. Instructions such as MADD.fmt were usually unfused, but were occasionally fused. Release 6 provides consistent behavior: MADDF.fmt and MSUBF.fmt are required to be fused in all implementations.

Release 6 MSUBF.fmt, `fd←fd-fs×ft`, corresponds more closely to pre-Release 6 NMADD.fmt, `fd←fr-fs×ft`, than to pre-Release 6 MSUB.fmt, `fd←fs×ft-fr`.

FPU scalar MADDF.fmt corresponds to MSA vector MADD.df.

FPU scalar MSUBF.fmt corresponds to MSA vector MSUB.df.

Operation:

```
if not IsCoprocessorEnabled(1)
   then SignalException(CoprocessorUnusable, 1) end if
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
if not IsFloatingPointImplemented(fmt))
   then SignalException(ReservedInstruction) end if
if fmt=D and FIR.D=0
   then SignalFPException(UnimplementedOperation) end if

vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vfd ← ValueFPR(fd, fmt)
MADDF.fmt: vinf ← vfd +∞ (vfs *∞ vft)
MADDF.fmt: vinf ← vfd -∞ (vfs *∞ vft)
StoreFPR(fd, fmt, vinf)
```

Special Considerations:

The fused multiply-add computation is performed in infinite precision, and signals Inexact, Overflow, or Underflow if and only if the final result differs from the infinite precision result in the appropriate manner.

Like most FPU computational instructions, if the flush-subnormals-to-zero mode, FCSR.FS=1, then subnormals are flushed before beginning the fused-multiply-add computation, and Inexact may be signaled.

I.e. Inexact may be signaled both by input flushing and/or by the fused-multiply-add: the conditions or ORed.

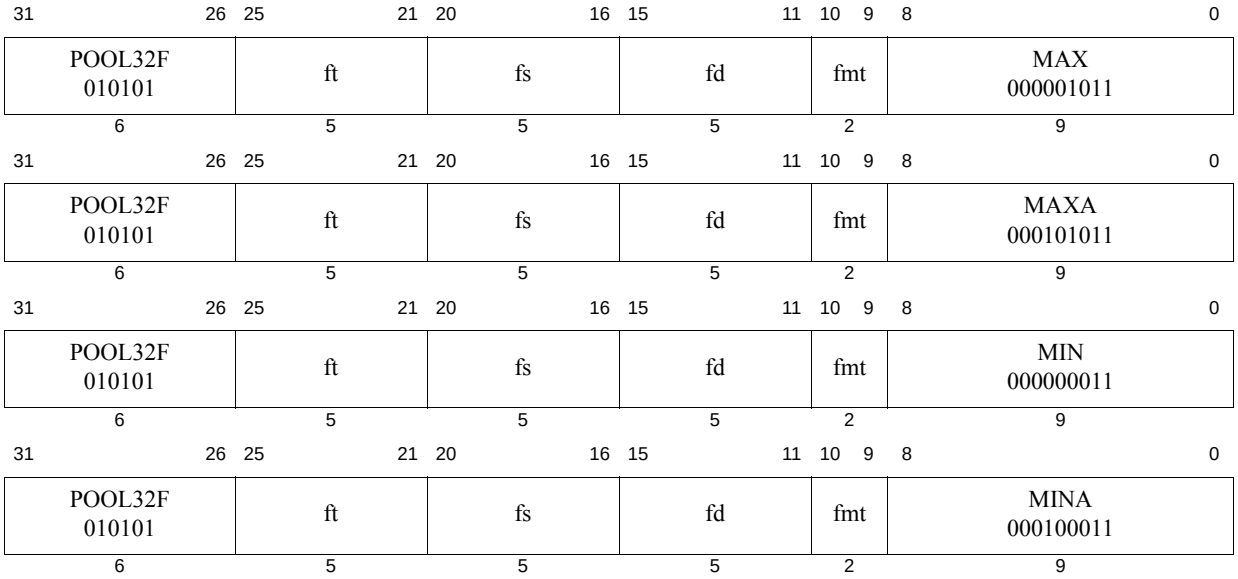
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `MAX.fmt MIN.fmt MAXA.fmt MINA.fmt`

`MAX.S fd,fs,ft`	**microMIPS32 Release 6**
`MAX.D fd,fs,ft`	**microMIPS32 Release 6**
`MAXA.S fd,fs,ft`	**microMIPS32 Release 6**
`MAXA.D fd,fs,ft`	**microMIPS32 Release 6**
`MIN.S fd,fs,ft`	**microMIPS32 Release 6**
`MIN.D fd,fs,ft`	**microMIPS32 Release 6**
`MINA.S fd,fs,ft`	**microMIPS32 Release 6**
`MINA.D fd,fs,ft`	**microMIPS32 Release 6**

Purpose: Scalar Floating-Point Max/Min/maxNumMag/minNumMag

Scalar Floating-Point Maximum

Scalar Floating-Point Minimum

Scalar Floating-Point argument with Maximum Absolute Value

Scalar Floating-Point argument with Minimum Absolute Value

Description:

```
MAX.fmt:   FPR[fd]← maxNum(FPR[fs],FPR[ft])
MIN.fmt:   FPR[fd]← minNum(FPR[fs],FPR[ft])
MAXA.fmt:  FPR[fd]← maxNumMag(FPR[fs],FPR[ft])
MINA.fmt:  FPR[fd]← minNumMag(FPR[fs],FPR[ft])
```

MAX.fmt writes the maximum value of the inputs `fs` and `ft` to the destination `fd`.

MIN.fmt writes the minimum value of the inputs `fs` and `ft` to the destination `fd`.

MAXA.fmt takes input arguments `fs` and `ft` and writes the argument with the maximum absolute value to the destination `fd`.

MINA.fmt takes input arguments `fs` and `ft` and writes the argument with the minimum absolute value to the destination `fd`.

The instructions MAX.fmt/MIN.fmt/MAXA.fmt/MINA.fmt correspond to the IEEE 754-2008 operations maxNum/

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

minNum/maxNumMag/minNumMag.

- MAX.fmt corresponds to the IEEE 754-2008 operation maxNum.
- MIN.fmt corresponds to the IEEE 754-2008 operation minNum.
- MAXA.fmt corresponds to the IEEE 754-2008 operation maxNumMag.
- MINA.fmt corresponds to the IEEE 754-2008 operation minNumMag.

Numbers are preferred to NaNs: if one input is a NaN, but not both, the value of the numeric input is returned. If both are NaNs, the NaN in `fs` is returned.[1]

The scalar FPU instructions MAX.fmt/MIN.fmt/MAXA.fmt/MINA.fmt correspond to the MSA instructions FMAX.df/FMIN.df/FMAXA.df/FMINA.df.

- Scalar FPU instruction MAX.fmt corresponds to the MSA vector instruction FMAX.df.
- Scalar FPU instruction MIN.fmt corresponds to the MSA vector instruction FMIN.df.
- Scalar FPU instruction MAXA.fmt corresponds to the MSA vector instruction FMAX_A.df.
- Scalar FPU instruction MINA.fmt corresponds to the MSA vector instruction FMIN_A.df.

Restrictions:

Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008. See also the section “Special Cases”, below.

Availability and Compatibility:

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

Operation:

```
if not IsCoprocessorEnabled(1)
   then SignalException(CoprocessorUnusable, 1) endif
if not IsFloatingPointImplemented(fmt)
   then SignalException(ReservedInstruction) endif

v1 ← ValueFPR(fs,fmt)
v2 ← ValueFPR(ft,fmt)

if SNaN(v1) or SNaN(v2) then
   then SignalException(InvalidOperand) endif

if NaN(v1) and NaN(v2)then
   ftmp ← v1
elseif NaN(v1) then
   ftmp ← v2
elseif NaN(v2) then
   ftmp ← v1
else
   case instruction of
```

1. IEEE standard 754-2008 allows either input to be chosen if both inputs are NaNs. Release 6 specifies that the first input must be propagated.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
    FMAX.fmt:   ftmp ← MaxFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
    FMIN.fmt:   ftmp ← MinFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
    FMAXA.fmt:  ftmp ← MaxAbsoluteFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
    FMINA.fmt:  ftmp ← MinAbsoluteFP.fmt(ValueFPR(fs,fmt),ValueFPR(ft,fmt))
    end case
endif

StoreFPR (fd, fmt, ftmp)
/* end of instruction */

function MaxFP(tt, ts, n)
    /* Returns the largest argument. */
endfunction MaxFP

function MinFP(tt, ts, n)
    /* Returns the smallest argument. */
endfunction MaxFP

function MaxAbsoluteFP(tt, ts, n)
    /* Returns the argument with largest absolute value.
       For equal absolute values, returns the largest positive argument.*/
endfunction MinAbsoluteFP

function MinAbsoluteFP(tt, ts, n)
    /* Returns the argument with smallest absolute value.
       For equal absolute values, returns the smallest positive argument.*/
endfunction MinAbsoluteFP

function NaN(tt, ts, n)
    /* Returns true if the value is a NaN */
    return SNaN(value) or QNaN(value)
endfunction MinAbsoluteFP
```

Table 5.26 Special Cases for FP MAX, MIN, MAXA, MINA

Operand		Other	Release 6 Instructions			
fs	ft		MAX	MIN	MAXA	MINA
-0.0	0.0		0.0	-0.0	0.0	-0.0
0.0	-0.0					
QNaN	#		#	#	#	#
#	QNaN					
QNaN1	QNaN2	Release 6	QNan1	QNaN1	QNaN1	QNaN1
		IEEE 754 2008	Arbitrary choice. Not allowed to clear sign bit.			

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 5.26 Special Cases for FP MAX, MIN, MAXA, MINA

Operand		Other	Release 6 Instructions			
fs	ft		MAX	MIN	MAXA	MINA
Either or both operands SNaN		Invalid Operation exception enabled	Signal Invalid Operation Exception. Destination not written.			
		... disabled	Treat as if the SNaN were a QNaN (do not quieten the result).			

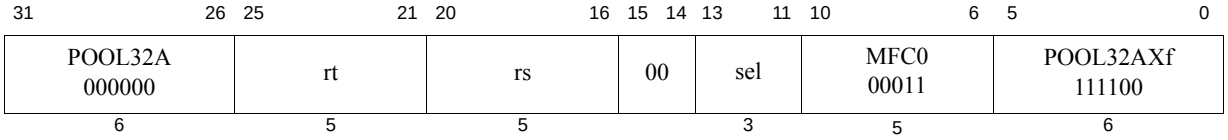
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `MFC0 rt, rs` **microMIPS**

`MFC0 rt, rs, sel` **microMIPS**

Purpose: Move from Coprocessor 0

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

Description: `GPR[rt] ← CPR[0,rs,sel]`

The contents of the coprocessor 0 register specified by the combination of *rs* and *sel* are loaded into general register *rt*. 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*.

Operation:

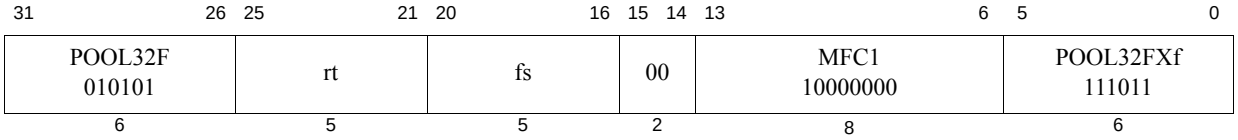
```
reg = rs
data ← CPR[0,reg,sel]
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← FPR[fs]`

The contents of FPR *fs* are loaded into general register *rt*.

Restrictions:

Operation:

```
data ← ValueFPR(fs, UNINTERPRETED_WORD)
GPR[rt] ← data
```

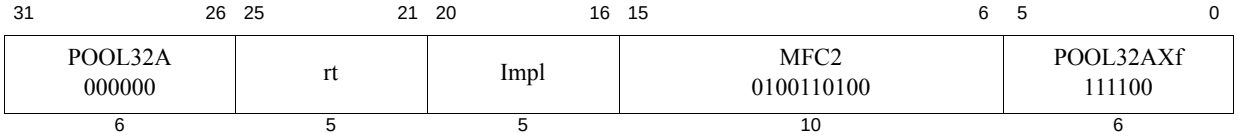
Exceptions:

Coprocessor Unusable, Reserved Instruction

Historical Information:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← CP2CPR[Impl]`

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

Restrictions:

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

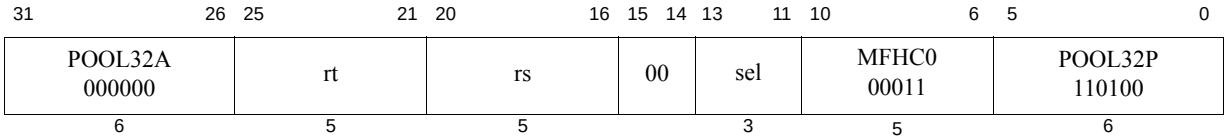
Operation:

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

Exceptions:

Coprocessor Unusable

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `MFHC0 rt, rs` **microMIPS Release 5**

`MFHC0 rt, rs, sel` **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] ← CPR[0,rs,sel][63:32]`

The contents of the Coprocessor 0 register specified by the combination of *rs* and *sel* are loaded into general register *rt*. Not all Coprocessor 0 registers support the *sel* field, and 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 the register exists but is not extended by 32-bits,or the register is extended for XPA, but XPA is not supported or enabled.

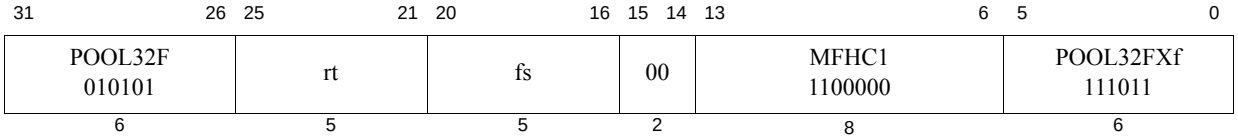
Operation:

```
reg ← rs
   data ← CPR[0,reg,sel]
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← FPR[fs]`$_{63..32}$

The contents of the high word of FPR *fs* are 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:

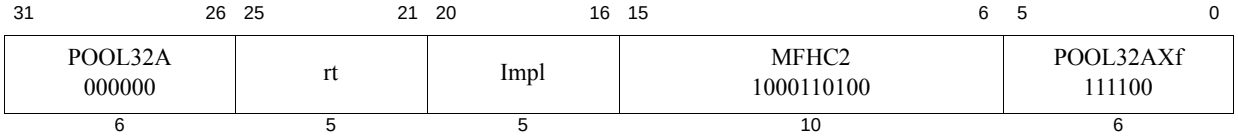
```
data ← ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)63..32
GPR[rt] ← data
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

The contents of the high word of the coprocessor 2 register denoted by the *Impl* field are 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 the *Impl* field 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:

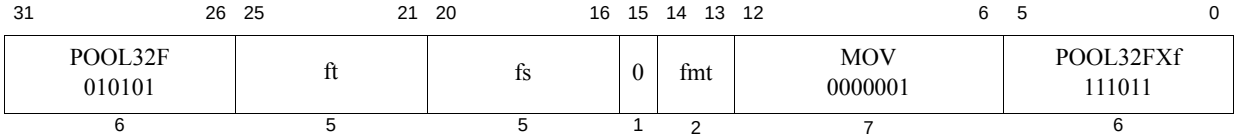
```
data ← CP2CPR[Impl]63..32
GPR[rt] ← data
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `MOV.fmt`

`MOV.S ft, fs` **microMIPS**

`MOV.D ft, fs` **microMIPS**

Purpose: Floating Point Move

To move an FP value between FPRs.

Description: `FPR[ft] ← FPR[fs]`

The value in FPR *fs* is placed into FPR *ft*. The source and destination are values in format *fmt*. In paired-single format, both the halves of the pair are copied to *ft*.

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 the fields are not valid, the result is **UNPREDICTABLE**.

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

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

Availability and Compatibility:

Not applicable.

Operation:

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

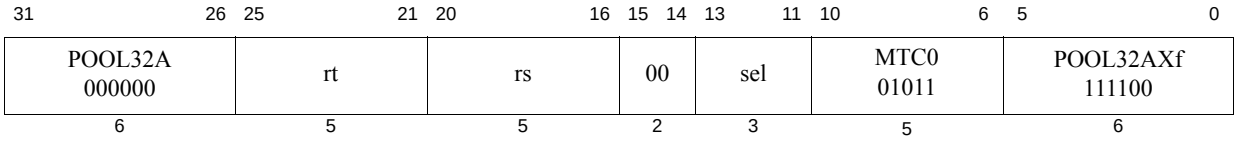
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `MTC0 rt, rs` **microMIPS**

`MTC0 rt, rs, sel` **microMIPS**

Purpose: Move to Coprocessor 0

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

Description: `CPR[0, rs, sel] ← 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.

Restrictions:

The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *rs* and *sel*.

Operation:

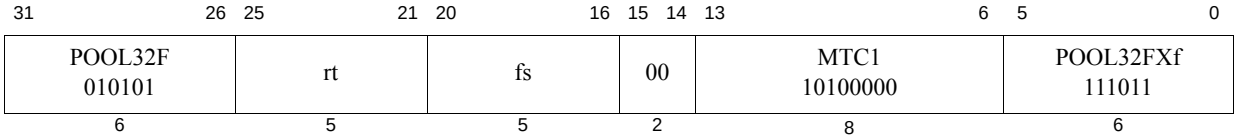
```
data ← GPR[rt]
reg ← rs
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Purpose: Move Word to Floating Point

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

Description: `FPR[fs] ← GPR[rt]`

The low word in GPR *rt* is placed into the low word of FPR *fs*.

Restrictions:

Operation:

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

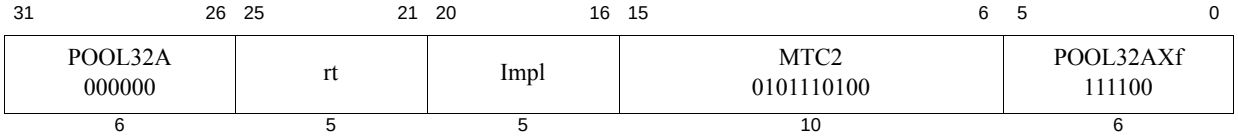
Exceptions:

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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← 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. 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 the *Impl* field specifies a Coprocessor 2 register that does not exist.

Operation:

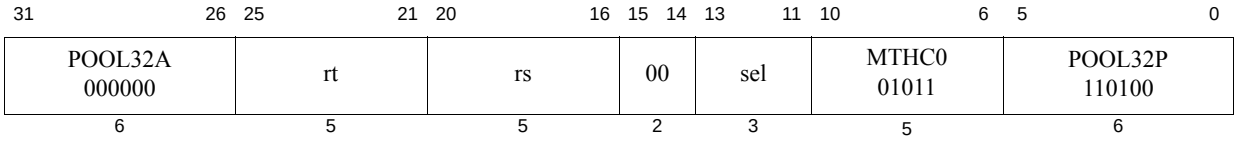
```
data ← GPR[rt]
CP2CPR[Impl] ← data
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `MTHC0 rt, rs` **microMIPS Release 5**

`MTHC0 rt, rs, sel` **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] ← 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; the *sel* field must be set to zero.

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.

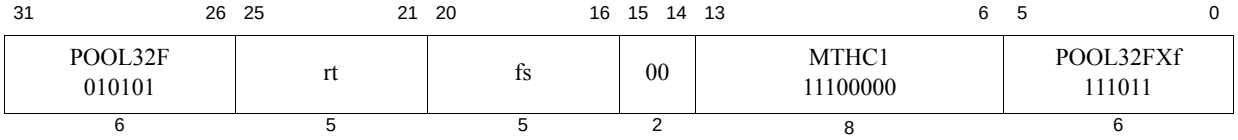
Operation:

```
data ← GPR[rt]
reg ← rs
CPR[0,reg,sel][63:32] ← data
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

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

Exceptions:

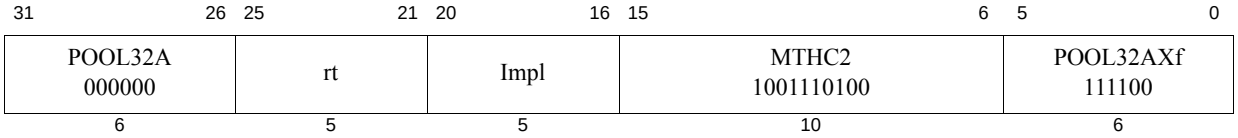
Coprocessor Unusable

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 is using an MTHC1 instruction to complete the operation, and sets the upper half of the 64-bit FPR to an **UNPREDICTABLE** value.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

The 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 the *Impl* field 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]
CP2CPR[Impl] ← data || CPR[2,rd,sel]31..0
```

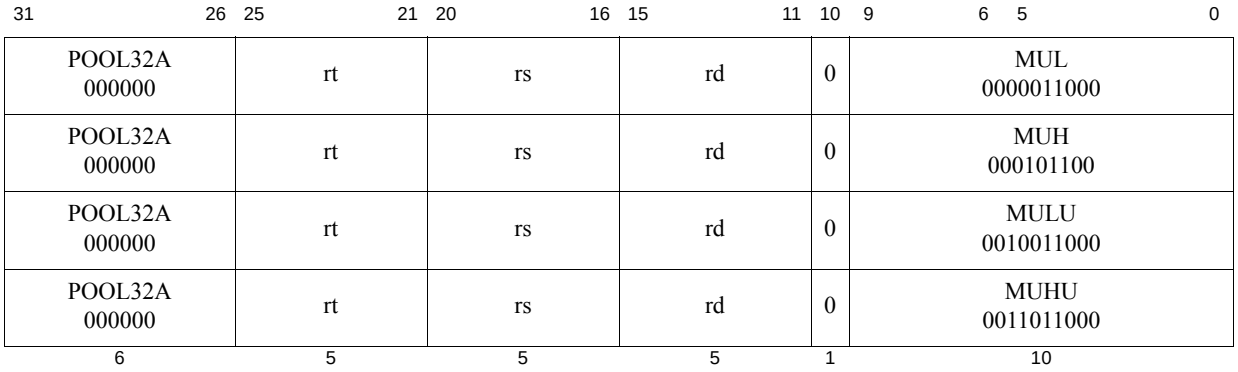
Exceptions:

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 is using an MTHC2 instruction to complete the operation, and sets the upper half of the 64-bit CPR to an **UNPREDICTABLE** value.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



31–26	25–21	20–16	15–11	10	9–6	5–0
POOL32A 000000	rt	rs	rd	0	MUL 0000011000	
POOL32A 000000	rt	rs	rd	0	MUH 000101100	
POOL32A 000000	rt	rs	rd	0	MULU 0010011000	
POOL32A 000000	rt	rs	rd	0	MUHU 0011011000	
6	5	5	5	1	10	

Format: `MUL MUH MULU MUHU`

`MUL rd,rs,rt` **microMIPS32 Release 6**

`MUH rd,rs,rt` **microMIPS32 Release 6**

`MULU rd,rs,rt` **microMIPS32 Release 6**

`MUHU rd,rs,rt` **microMIPS32 Release 6**

Purpose: Multiply Integers (with result to GPR)

MUL: Multiply Words Signed, Low Word
MUH: Multiply Words Signed, High Word
MULU: Multiply Words Signed, Low Word
MUHU: Multiply Words Signed, High Word

Description:

```
MUL:   GPR[rd] ← sign_extend.32( lo_word( multiply.signed( GPR[rs] × GPR[rt] )
MUH:   GPR[rd] ← sign_extend.32( hi_word( multiply.signed( GPR[rs] × GPR[rt] )
MULU:  GPR[rd] ← sign_extend.32( lo_word( multiply.unsigned( GPR[rs] ×GPR[rt] )
MUHU:  GPR[rd] ← sign_extend.32( hi_word( multiply.unsigned( GPR[rs] ×GPR[rt] )
```

The Release 6 multiply instructions multiply the operands in GPR[rs] and GPR[rd], and place the specified high or low part of the result, of the same width, in GPR[rd].

MUL performs a signed 32-bit integer multiplication, and places the low 32 bits of the result in the destination register.

MUH performs a signed 32-bit integer multiplication, and places the high 32 bits of the result in the destination register.

MULU performs an unsigned 32-bit integer multiplication, and places the low 32 bits of the result in the destination register.

MUHU performs an unsigned 32-bit integer multiplication, and places the high 32 bits of the result in the destination register.

Restrictions:

MUL behaves correctly even if its inputs are not sign extended 32-bit integers. Bits 32-63 of its inputs do not affect the result.

MULU behaves correctly even if its inputs are not zero or sign extended 32-bit integers. Bits 32-63 of its inputs do not affect the result.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Availability and Compatibility:

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

Programming Notes:

The low half of the integer multiplication result is identical for signed and unsigned. Nevertheless, there are distinct instructions MUL MULU . Implementations may choose to optimize a multiply that produces the low half followed by a multiply that produces the upper half. Programmers are recommended to use matching lower and upper half multiplications.

Release 6 MUL instruction has the same opcode mnemonic as the pre-Release 6 MUL instruction. The semantics of these instructions are almost identical: both produce the low 32-bits of the 32×32=64 product; but the pre-Release 6 MUL is unpredictable if its inputs are not properly sign extended 32-bit values on a 64 bit machine, and is defined to render the HI and LO registers unpredictable, whereas the Release 6 version ignores bits 32-63 of the input, and there are no HI/LO registers in Release 6 to be affected. If disambiguation is necessary, say Release 6 versus pre-Release 6 or specify the instruction string.

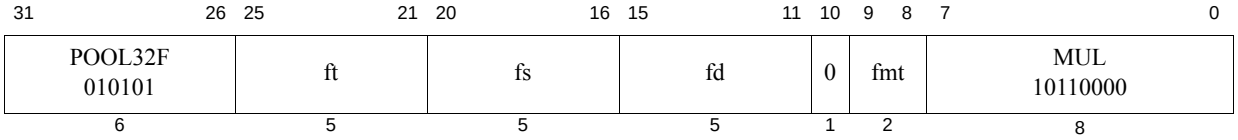
Operation:

```
MUH: if NotWordValue(GPR[rs]) then UNPREDICTABLE endif
MUH: if NotWordValue(GPR[rt]) then UNPREDICTABLE endif
MUHU: if not(zero_or_sign_extended.32(GPR[rs])) then UNPREDICTABLE endif
MUHU: if not(zero_or_sign_extended.32(GPR[rt])) then UNPREDICTABLE endif

/* recommended imnplementation: ignore bits 32-63 for MUL, MUH, MULU, MUHU */

MUL, MUH:
  s1 ← signed_word(GPR[rs])
  s2 ← signed_word(GPR[rt])
MULU, MUHU:
  s1 ← unsigned_word(GPR[rs])
  s2 ← unsigned_word(GPR[rt])

product ← s1 × s2     /* product is twice the width of sources */

MUL:   GPR[rd] ← sign_extend.32( lo_word( product )
MUH:   GPR[rd] ← sign_extend.32( hi_word( product )
MULU:  GPR[rd] ← sign_extend.32( lo_word( product )
MUHU:  GPR[rd] ← sign_extend.32( hi_word( product )
endif
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `MUL.fmt`

`MUL.S fd, fs, ft` **microMIPS**

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

Purpose: Floating Point Multiply

To multiply FP values.

Description: `FPR[fd] ← FPR[fs] x FPR[ft]`

The value in FPR *fs* is multiplied by the value in FPR *ft*. The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. MUL.PS multiplies the upper and lower halves of FPR *fs* and FPR *ft* independently, and ORs together any generated exceptional conditions.

Restrictions:

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

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

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

Availability and Compatibility:

Not applicable.

Operation:

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

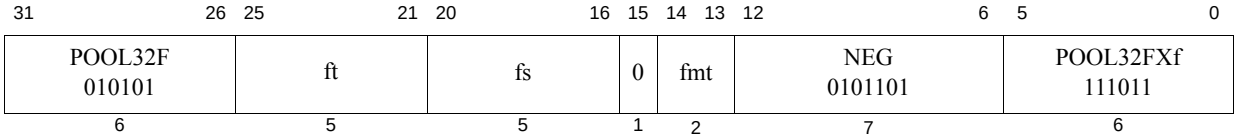
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `NEG.fmt`

`NEG.S ft, fs` **microMIPS**

`NEG.D ft, fs` **microMIPS**

Purpose: Floating Point Negate

To negate an FP value.

Description: `FPR[ft] ← -FPR[fs]`

The value in FPR *fs* is negated and placed into FPR *ft*. The value is negated by changing the sign bit value. The operand and result are values in format *fmt*. NEG.PS negates the upper and lower halves of FPR *fs* independently, and ORs together any generated exceptional conditions.

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

If $FCSR_{ABS2008}$=1 then this operation is non-arithmetic. For this case, both regular floating point numbers and NAN values are treated alike, only the sign bit is affected by this instruction. No IEEE 754 exception can be generated for this case.

Restrictions:

The fields *fs* and *ft* must specify FPRs valid for operands of type *fmt*. If the fields are not valid, the result is **UNPREDICTABLE**. The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

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

Operation:

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

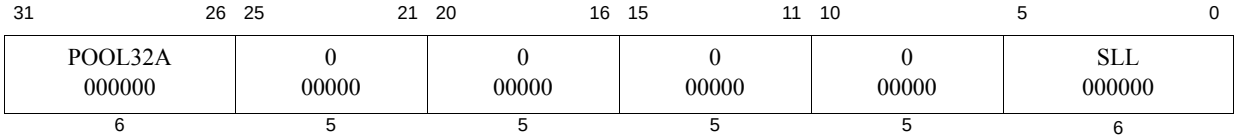
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Operations:

None

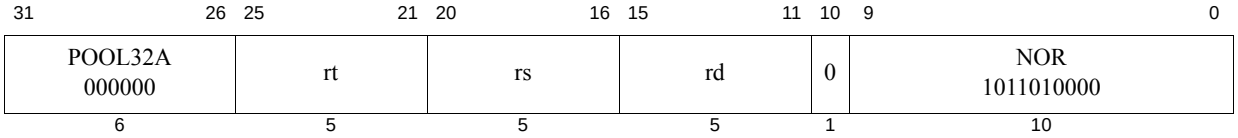
Exceptions:

None

Programming Notes:

The zero instruction word, which represents SLL, r0, r0, 0, is the preferred NOP for software to use and to pad out alignment sequences.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `NOR rd, rs, rt` **microMIPS**

Purpose: Not Or

To do a bitwise logical NOT OR.

Description: `GPR[rd] ← GPR[rs] NOR GPR[rt]`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical NOR operation. The result is placed into GPR *rd*.

Restrictions:

None

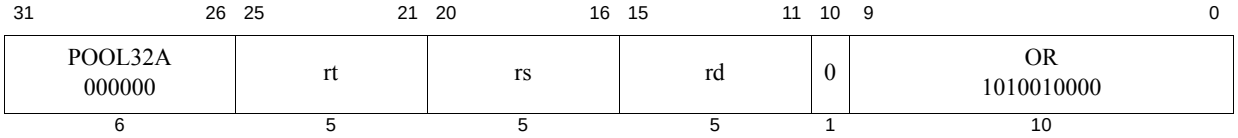
Operation:

```
GPR[rd] ← GPR[rs] nor GPR[rt]
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `OR rd, rs, rt` **microMIPS**

Purpose: Or

To do a bitwise logical OR.

Description: `GPR[rd] ← GPR[rs] or GPR[rt]`

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

Restrictions:

None

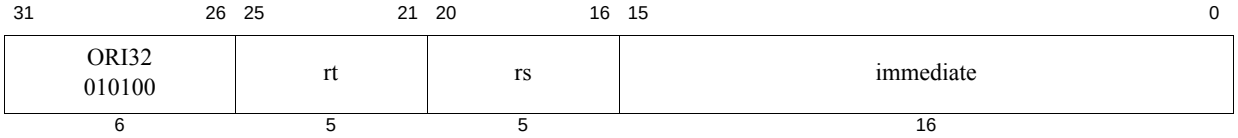
Operations:

```
GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `ORI rt, rs, immediate` **microMIPS**

Purpose: Or Immediate

To do a bitwise logical OR with a constant.

Description: `GPR[rt] ← GPR[rs] or immediate`

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

Restrictions:

None

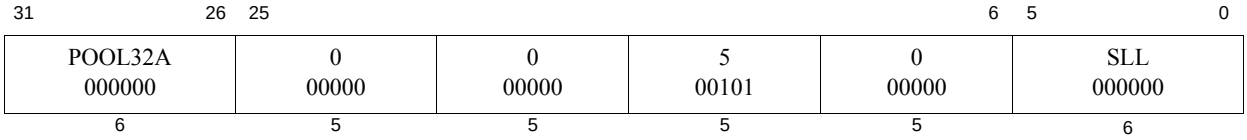
Operations:

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

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `PAUSE` **microMIPS**

Purpose: Wait for the LLBit to clear.

Description:

Locks implemented using the LL/SC instructions are a common method of synchronization between threads of control. A 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, 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 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:

Pre-Release 6: The operation of the processor is **UNPREDICTABLE** if a PAUSE instruction is executed placed in the delay slot of a branch or jump instruction.

Operations:

```
if LLBit ≠ 0 then
   EPC ← PC + 4                  /* Resume at the following instruction */
   DescheduleInstructionStream()
endif
```

Exceptions:

None

Programming Notes:

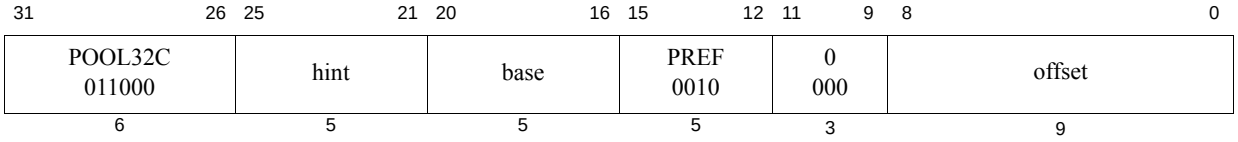
The PAUSE instruction is intended to be inserted into the instruction stream after an LL instruction has set the LLBit and found the software lock set. The program may wait forever if a PAUSE instruction is executed and there is no possibility that the LLBit will ever be cleared.

An example use of the PAUSE instruction is included in the following example:

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
acquire_lock:
   ll     t0, 0(a0)               /* Read software lock, set hardware lock */
   bnezc  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 */
   bnezc  t0, 10f                 /* Branch if lock acquired successfully */
   sync
acquire_lock_retry:
   pause                          /* Wait for LLBIT to clear before retry */
   bc     acquire_lock            /* and retry the operation */
10:

   Critical region code

release_lock:
   sync
   sw     zero, 0(a0)             /* Release software lock, clearing LLBIT */
                                  /* for any PAUSEd waiters */
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `PREF hint,offset(base)` **microMIPS**

Purpose: Prefetch

To move data between memory and cache.

Description: `prefetch_memory(GPR[base] + offset)`

PREF adds the 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 6.27 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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 6.27 Values of *hint* Field for PREF Instruction (Continued)

Value	Name	Data Use and Desired Prefetch Action
1	store	Use: Prefetched data is expected to be stored or modified. Action: Fetch data as if for a store.
2	L1 LRU hint	Pre-Release 6: Reserved for Architecture. Implementation dependent in Release 6. This hint code marks the line as LRU in the L1 cache and thus preferred for next eviction. Implementations can choose to writeback and/or invalidate as long as no architectural state is modified.
3	Reserved	Pre-Release 6: Reserved for Architecture. Release 6: Available for implementation dependent use.
4	load_streamed	Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache. Action: Fetch data as if for a load and place it in the cache so that it does not displace data prefetched as “retained.”
5	store_streamed	Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache. Action: Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as “retained.”
6	load_retained	Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as “streamed.”
7	store_retained	Use: Prefetched data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a store and place it in the cache so that it is not displaced by data prefetched as “streamed.”
8-15	Reserved	In the Release 6 architecture, hint codes 8 - 15 are treated the same as hint codes 0 - 7 respectively, but operate on the L2 cache.
16-23	Reserved	In the Release 6 architecture, hint codes 16 - 23 are treated the same as hint codes 0 - 7 respectively, but operate on the L3 cache.
24	Reserved	Pre-Release 6: Unassigned by the Architecture - available for implementation-dependent use. This hint code is not implemented in the Release 6 architecture and generates a Reserved Instruction exception (RI).
25	writeback_invalidate (also known as “nudge”) Reserved in Release 6	Use: Data is no longer expected to be used. Action: For a writeback cache, schedule a writeback of any dirty data. At the completion of the writeback, mark the state of any cache lines written back as invalid. If the cache line is not dirty, it is implementation dependent whether the state of the cache line is marked invalid or left unchanged. If the cache line is locked, no action is taken. This hint code is not implemented in the Release 6 architecture and generates a Reserved Instruction exception (RI).
26-29	Reserved	Pre-Release 6: Unassigned by the Architecture — available for implementation-dependent use. These hints are not implemented in the Release 6 architecture and generate a Reserved Instruction exception (RI).

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 6.27 Values of *hint* Field for PREF Instruction (Continued)

Value	Name	Data Use and Desired Prefetch Action
30	PrepareForStore Reserved in Release 6	Use: Prepare the cache for writing an entire line, without the overhead involved in filling the line from memory. Action: If the reference hits in the cache, no action is taken. If the reference misses in the cache, a line is selected for replacement, any valid and dirty victim is written back to memory, the entire line is filled with zero data, and the state of the line is marked as valid and dirty. Programming Note: Because the cache line is filled with zero data on a cache miss, software must not assume that this action, in and of itself, can be used as a fast bzero-type function. This hint is not implemented in the Release 6 architecture and generates a Reserved Instruction exception (RI).
31	Reserved.	Pre-Release 6: Unassigned by the Architecture — available for implementation-dependent use. This hint is not implemented in the Release 6 architecture and generates a Reserved Instruction exception (RI).

Restrictions:

None

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

Availability and Compatibility:

This instruction has been reallocated an opcode in Release 6.

Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Bus Error, Cache Error

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:

In the Release 6 architecture, hint codes 2:3, 10:11, 18:19 behave as a NOP if not implemented. Hint codes 24:31 are not implemented (treated as reserved) and always signal a Reserved Instruction exception (RI).

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

Prefetch cannot move data to or from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. A prefetch may be used using an address pointer before the validity of the pointer is determined without worrying about an addressing exception.

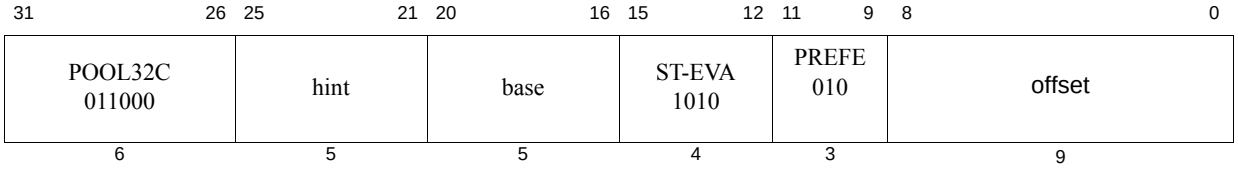
It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREF instruction. Typically, this only occurs in systems which have high-reliability requirements.

Prefetch operations have no effect on cache lines that were previously locked with the CACHE instruction.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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, 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 (for example, kseg1), the programmed cacheability and coherency attribute of a segment (for example, 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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 6.28 Values of *hint* Field for PREFE Instruction

Value	Name	Data Use and Desired Prefetch Action
0	load	Use: Prefetched data is expected to be read (not modified). Action: Fetch data as if for a load.
1	store	Use: Prefetched data is expected to be stored or modified. Action: Fetch data as if for a store.
2	Reserved Implementation Dependent	Reserved for future use - not available to implementations. Implementation dependent in Release 6. This hint code marks the line as LRU in the L1 cache and thus preferred for next eviction. Implementations can choose to writeback and/or invalidate as long as no architectural state is modified.
3	Reserved Implementation Dependent	Reserved for future use - not available to implementations. This hint code is unused In the Release 6 architecture.
4	load_streamed	Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache. Action: Fetch data as if for a load and place it in the cache so that it does not displace data prefetched as “retained.”
5	store_streamed	Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache. Action: Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as “retained.”
6	load_retained	Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as “streamed.”
7	store_retained	Use: Prefetched data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a store and place it in the cache so that it is not displaced by data prefetched as “streamed.”
8-15	Reserved	In the Release 6 architecture, hint codes 8 - 15 are treated the same as hint codes 0 - 7 respectively, but operate on the L2 cache.
16-23	Reserved	In the Release 6 architecture, hint codes 16 - 23 are treated the same as hint codes 0 - 7 respectively, but operate on the L3 cache.
8-20	Reserved	Reserved for future use - not available to implementations.
21-23	Implementation Dependent	Unassigned by the Architecture - available for implementation-dependent use.
24	Implementation Dependent Reserved in Release 6	Unassigned by the Architecture - available for implementation-dependent use. This hint code is not implemented in the Release 6 architecture and generates a Reserved Instruction exception (RI).

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 6.28 Values of *hint* Field for PREFE Instruction (Continued)

Value	Name	Data Use and Desired Prefetch Action
25	writeback_invalidate (also known as “nudge”) Reserved in Release 6	Use: Data is no longer expected to be used. Action: For a writeback cache, schedule a writeback of any dirty data. At the completion of the writeback, mark the state of any cache lines written back as invalid. If the cache line is not dirty, it is implementation dependent whether the state of the cache line is marked invalid or left unchanged. If the cache line is locked, no action is taken. This hint code is not implemented in the Release 6 architecture and generates a Reserved Instruction exception (RI).
26-29	Implementation Dependent Reserved in Release 6	Unassigned by the Architecture - available for implementation-dependent use. These hint codes are not implemented in the Release 6 architecture and generate a Reserved Instruction exception (RI).
30	PrepareForStore Reserved in Release 6	Use: Prepare the cache for writing an entire line, without the overhead involved in filling the line from memory. Action: If the reference hits in the cache, no action is taken. If the reference misses in the cache, a line is selected for replacement, any valid and dirty victim is written back to memory, the entire line is filled with zero data, and the state of the line is marked as valid and dirty. Programming Note: Because the cache line is filled with zero data on a cache miss, software must not assume that this action, in and of itself, can be used as a fast bzero-type function. This hint code is not implemented in the Release 6 architecture and generates a Reserved Instruction exception (RI).
31	Implementation Dependent Reserved in Release 6	Unassigned by the Architecture - available for implementation-dependent use. This hint code is not implemented in the Release 6 architecture and generates a Reserved Instruction exception (RI).

Restrictions:

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

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

Operation:

```
vAddr ← GGPR[base] + sign_extend(offset)
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Bus Error, Cache Error, Address Error, Reserved Instruction, Coprocessor Usable

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:

In the Release 6 architecture, hint codes 0:23 behave as a NOP and never signal a Reserved Instruction exception (RI). Hint codes 24:31 are not implemented (treated as reserved) and always signal a Reserved Instruction exception (RI).

Prefetch cannot move data to or from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

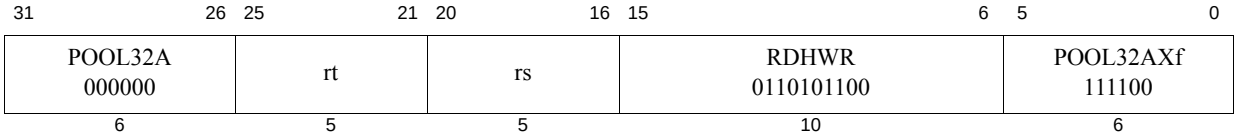
Prefetch does not cause addressing exceptions. A prefetch may be used using an address pointer before the validity of the pointer is determined without worrying about an addressing exception.

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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← HWR[rs]`

If access is allowed to the specified hardware register, the contents of the register specified by *rs* is 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 6.29.

Table 6.29 RDHWR Register Numbers

Register Number (*rd* Value)	Mnemonic	Description
0	CPUNum	Number of the CPU on which the program is currently running. This register provides read access to the coprocessor 0 $EBase_{CPUNum}$ field.
1	SYNCI_Step	Address step size to be used with the SYNCI instruction, or zero if no caches need be synchronized. See that instruction's description for the use of this value.
2	CC	High-resolution cycle counter. This register provides read access to the coprocessor 0 *Count* Register.
3	CCRes	Resolution of the CC register. This value denotes the number of cycles between update of the register. For example: CCRes Value / Meaning 1 — CC register increments every CPU cycle 2 — CC register increments every second CPU cycle 3 — CC register increments every third CPU cycle etc.
4-28		These registers numbers are reserved for future architecture use. Access results in a Reserved Instruction Exception.
29	ULR	User Local Register. This register provides read access to the coprocessor 0 *UserLocal* register, if it is implemented. In some operating environments, the *UserLocal* register is a pointer to a thread-specific storage block.
30-31		These register numbers are reserved for implementation-dependent use. If they are not implemented, access results in a Reserved Instruction Exception.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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.

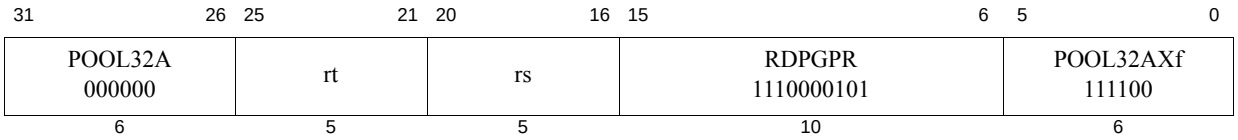
Operation:

```
case rs
       0: temp ← EBase_CPUNum
   1: temp ← SYNCI_StepSize()
   2: temp ← Count
   3: temp ← CountResolution()
   29: temp ← UserLocal
   30: temp ← Implementation-Dependent-Value
   31: temp ← Implementation-Dependent-Value
   otherwise: SignalException(ReservedInstruction)
endcase
GPR[rt] ← temp
```

Exceptions:

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← 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:

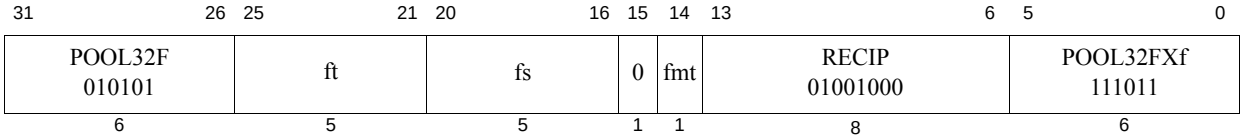
```
GPR[rt] ← SGPR[SRSCtlPSS, rs]
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `RECIP.fmt`

`RECIP.S ft, fs` **microMIPS**

`RECIP.D ft, fs` **microMIPS**

Purpose: Reciprocal Approximation

To approximate the reciprocal of an FP value (quickly).

Description: `FPR[ft] ← 1.0 / FPR[fs]`

The reciprocal 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 the both the exact result and the IEEE-mandated representation of the exact result by no more than one unit in the least-significant place (ULP).

It is implementation dependent whether the result is affected by the current rounding mode in *FCSR*.

Restrictions:

The fields *fs* and *ft* must specify FPRs valid for operands of type *fmt. I*f the fields 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**.

Availability and Compatibility:

RECIP.S and RECIP.D: Required in all versions of MIPS64 since MIPS64 Release 1. Not available in MIPS32 Release 1. Required in MIPS32 Release 2 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:

```
StoreFPR(ft, fmt, 1.0 / valueFPR(fs, fmt))
```

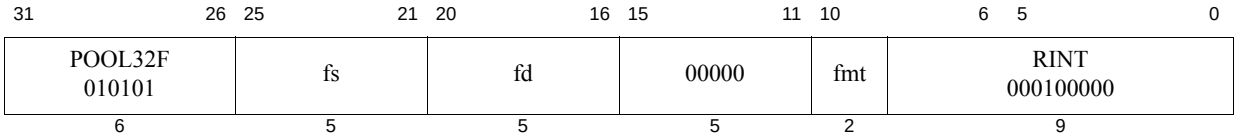
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `RINT.fmt`
`RINT fd, fs` **microMIPS32 Release 6**

Purpose: Floating-Point Round to Integral

Scalar floating-point round to integral floating point value.

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

The scalar floating-point value in the register `fs` is rounded to an integral valued floating-point number in the same format based on the rounding mode bits RM in the FPU Control and Status Register *FCSR*. The result is written to `fd`.

The operands and results are values in floating-point data format *fmt*.

The RINT.fmt instruction corresponds to the **roundToIntegralExact** operation in the IEEE Standard for Floating-Point Arithmetic 754™-2008. The Inexact exception is signaled if the result does not have the same numerical value as the input operand.

The floating point scalar instruction RINT.fmt corresponds to the MSA vector instruction FRINT.df. I.e. RINT.S corresponds to FRINT.W, and RINT.D corresponds to FRINT.D.

Restrictions:

Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

Availability and Compatibility:

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

Operation:

RINT.fmt:

```
if not IsCoprocessorEnabled(1)
   then SignalException(CoprocessorUnusable, 1) end if
if not IsFloatingPointImplemented(fmt))
   then SignalException(ReservedInstruction) end if
if fmt=D and FIR.D=0
   then SignalFPException(UnimplementedOperation) end if

fin ← ValueFPR(fs,fmt)
ftmp ←RoundIntFP(fin, fmt)
if( fin ≠ ftmp ) SignalFPException(InExact)
StoreFPR (fd, fmt, ftmp )

function RoundIntFP(tt, n)
   /* Round to integer operation, using rounding mode FCSR.RM*/
endfunction RoundIntFP
```

Exceptions:

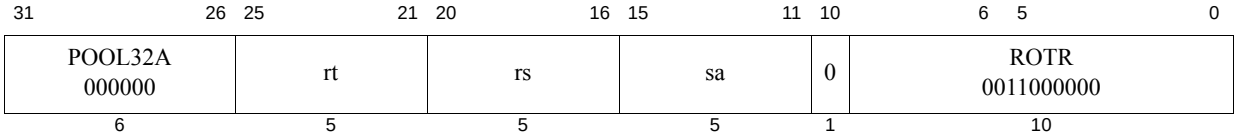
Coprocessor Unusable, Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[rs] ×(right) sa`

The contents of the low-order 32-bit word of GPR *rs* are rotated right; the word result is placed in GPR *rt*. The bit-rotate amount is specified by *sa*.

Restrictions:

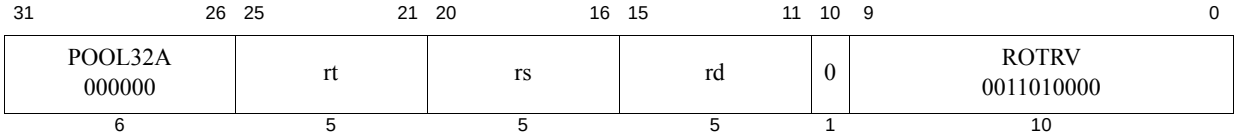
Operation:

```
if ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
   UNPREDICTABLE
endif
s ← sa
temp ← GPR[rs]s-1..0 || GPR[rs]31..s
GPR[rt] ← temp
```

Exceptions:

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[rt] ×(right) GPR[rs]`

The contents of the low-order 32-bit word of GPR *rt* are rotated right; the word result is placed in GPR *rd*. The bit-rotate amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

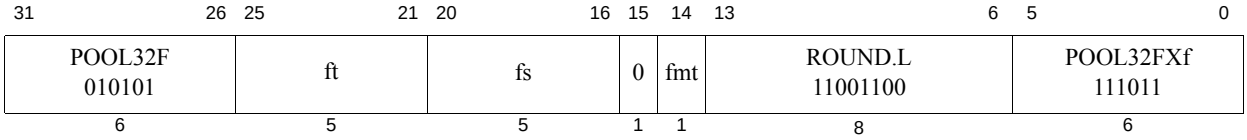
Operation:

```
if ((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] ← temp
```

Exceptions:

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `ROUND.L.fmt`

`ROUND.L.S ft, fs` **microMIPS**

`ROUND.L.D ft, fs` **microMIPS**

Purpose: Floating Point Round to Long Fixed Point

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

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

The value in FPR *fs*, in format *fmt*, is converted to a value in 64-bit long fixed point format and rounded to nearest/even (rounding mode 0). The result is placed in FPR *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 and an IEEE Invalid Operation condition exists. 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, $2^{63}-1$, is written to *ft*.

Restrictions:

The fields *fs* and *ft* must specify valid FPRs: *fs* for type *fmt* and *fd* for long fixed point. If the fields 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))
```

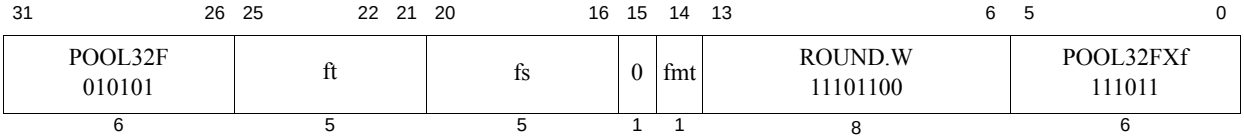
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `ROUND.W.fmt`

`ROUND.W.S ft, fs` **microMIPS**

`ROUND.W.D ft, fs` **microMIPS**

Purpose: Floating Point Round to Word Fixed Point

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

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

The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format rounding to nearest/even (rounding mode 0). The result is placed in FPR *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. 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 the fields 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))
```

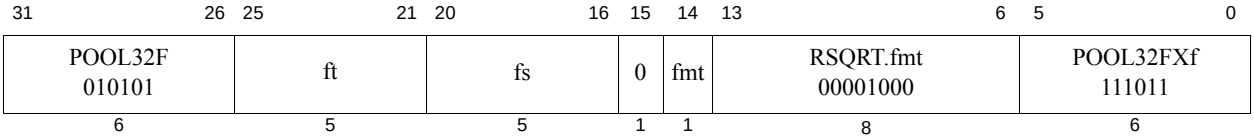
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `RSQRT.fmt`

`RSQRT.S ft, fs` **microMIPS**

`RSQRT.D ft, fs` **microMIPS**

Purpose: Reciprocal Square Root Approximation

To approximate the reciprocal of the square root of an FP value (quickly).

Description: `FPR[ft] ← 1.0 / 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 the fields 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**.

Availability and Compatibility:

RSQRT.S and RSQRT.D: Required in all versions of MIPS64 since MIPS64 Release 1. Not available in MIPS32 Release 1. Required in MIPS32 Release 2 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:

```
StoreFPR(ft, fmt, 1.0 / SquareRoot(valueFPR(fs, fmt)))
```

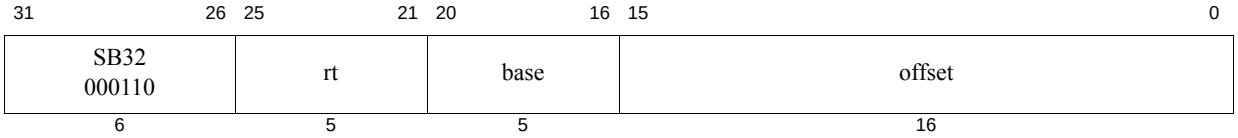
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SB rt, offset(base)` **microMIPS**

Purpose: Store Byte

To store a byte to memory.

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

The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

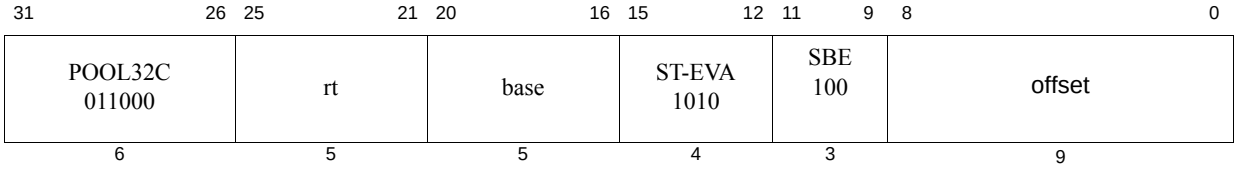
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1.. || (pAddr..0 xor ReverseEndian)
bytesel ← vAddr..0 xor BigEndianCPU
dataword ← GPR[rt]_-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The SBE instruction functions the same 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 1.

Restrictions:

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

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1.. || (pAddr_..0 xor ReverseEndian)
bytesel ← vAddr_..0 xor BigEndianCPU
dataword ← GPR[rt]_-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
```

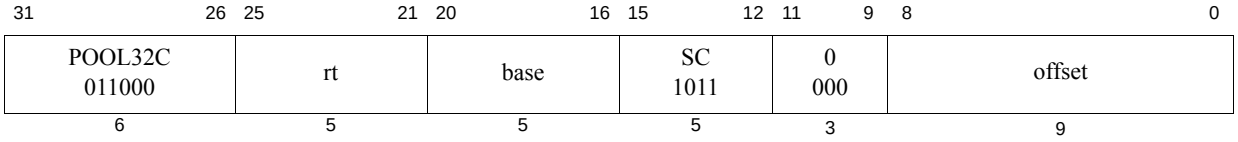
Exceptions:

TLB Refill, TLB Invalid

Bus Error, Address Error

Watch, Reserved Instruction, Coprocessor Unusable,

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SC rt, offset(base)` **microMIPS**

Purpose: Store Conditional Word

To store a word to memory to complete an atomic read-modify-write

Description: `if atomic_update then memory[GPR[base] + offset] ← GPR[rt], GPR[rt] ← 1 else GPR[rt] ← 0`

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations on synchronizable memory locations. In Release 5, the behavior of SC is modified when $Config5_{LLB}$=1.

The 32-bit word in GPR *rt* is conditionally stored in memory at the location specified by the aligned effective address. The signed *offset* is added to the contents of GPR *base* to form an effective address.

The SC completes the RMW sequence begun by the preceding LL instruction executed on the processor. To complete the RMW sequence atomically, the following occur:

- The 32-bit word of GPR *rt* is stored to memory at the location specified by the aligned effective address.
- A one, indicating success, is written into GPR *rt*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *rt*.

If either of the following events occurs between the execution of LL and SC, the SC fails:

- A coherent store is completed by another processor or coherent I/O module into the block of synchronizable physical memory containing the word. The size and alignment of the block is implementation-dependent, but it is at least one word and at most the minimum page size.
- A coherent store is executed between an LL and SC sequence on the same processor to the block of synchronizable physical memory containing the word (if $Config5_{LLB}$=1; else whether such a store causes the SC to fail is not predictable).
- An ERET instruction is executed. (Release 5 includes ERETNC, which will not cause the SC to fail.)

Furthermore, an SC must always compare its address against that of the LL. An SC will fail if the aligned address of the SC does not match that of the preceeding LL.

A load that executes on the processor executing the LL/SC sequence to the block of synchronizable physical memory containing the word, will not cause the SC to fail (if $Config5_{LLB}$=1; else such a load may cause the SC to fail).

If any of the events listed below occurs between the execution of LL and SC, the SC may fail where it could have succeeded, i.e., success is not predictable. Portable programs should not cause any of these events.

- A load or store executed on the processor executing the LL and SC that is not to the block of synchronizable physical memory containing the word. (The load or store may cause a cache eviction between the LL and SC that results in SC failure. The load or store does not necessarily have to occur between the LL and SC.)

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

- 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.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (AddressError) on a misaligned access is required.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
if LLbit then
   StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 0 || 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.

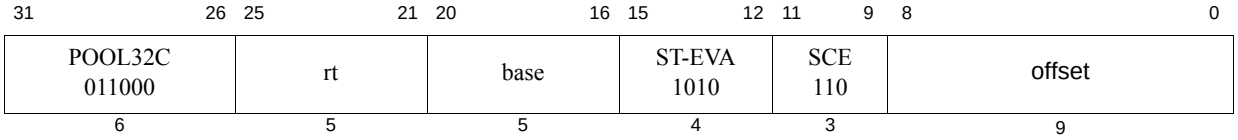
```
L1:
   LL     T1, (T0)  # load counter
   ADDI   T2, T1, 1 # increment
   SC     T2, (T0)  # try to store, checking for atomicity
   BEQC   T2, 0, L1 # if not atomic (0), try again
```

Exceptions between the LL and SC cause SC to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LL and SC function on a single processor for *cached noncoherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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 memory[GPR[base] + offset] ← GPR[rt], GPR[rt] ← 1 else GPR[rt] ← 0`

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

The 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 occurs:

- The 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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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 non coherent* 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 the same 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 1.

Restrictions:

The addressed location must have a memory access type of *cached non coherent* 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.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (AddressError) on a misaligned access is required.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
if LLbit then
   StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 0 || 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
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

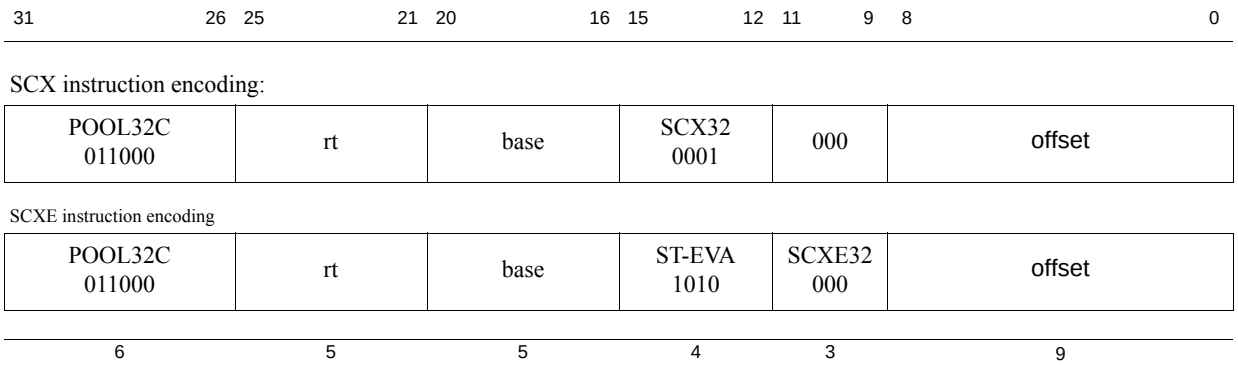
```
ADDI   T2, T1, 1 # increment
SCE    T2, (T0)  # try to store, checking for atomicity
BEQC   T2, 0, L1 # if not atomic (0), try again
```

Exceptions between the LLE and SCE cause SCE to fail, so persistent exceptions must be avoided. Examples 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 non coherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SCX, SCXE`

`SCX rt, offset(base)` **microMIPS32 Release 6**

`SCXE rt, offset(base)` **microMIPS32 Release 6**

Purpose: Store Conditional Extended {Word,Word EVA}

Store to memory as part of an extended LLX/LL-SCX/SC sequence; word, or word EVA

Description:

The LLX/SCX family of instructions (SCX, SCXE) extends the MIPS LL/SC mechanism for performing atomic read-modify-writes to permit more than one memory location to be written atomically. The memory locations are constrained to be aligned, adjacent and within both the same synchronization block and the same cache line (if applicable).

LL-SC code sequences in general, and LLX/LL-SCX/SC in particular, provide atomicity if the computer system can guarantee that, if the SC passes, then atomicity has not been violated by transactions between the LL and SC. It should also guarantee eventual success, i.e. that failures will not persist forever.

The signed *offset* is added to the contents of GPR *base* to form an effective address. This address must be naturally aligned.

An SCX/SCXE instruction (at PC) must be followed by a matching SC/SCE instruction (at PC+4).

For SCX and SCXE the 32-bit word in GPR *rt* is concatenated with the 32-bit word of the following SC instruction's GPR *rt* to form the 64-bit doubleword data to be conditionally stored.

The SCX/SC family instruction double width store data is performed if it can be guaranteed that there has been no violation of atomicity since the preceding LLX/LL family instruction. If such atomicity cannot be guaranteed, then the conditional store fails. A value is written into the rt register of the SC family instruction that follows the SCX family instruction: 0 if failure, 1 if success.

If the following SC-family (SC, SCE) instruction succeeds, then the SCX-family instruction (SCX, SCXE) also succeeds, and the store data from both the SCX and SC are concatenated and committed to memory atomically as a double width transaction. If the SC fails, then the SCX also fails, and neither commit to memory. The SC instruction at PC+4 modifies a GPR to indicate success or failure of both the SC and SCX.

In particular, the SCX/SCXE and SC/SCE data addresses must be adjacent, within the same synchronization block, non-overlapping, and naturally-aligned appropriately (for a 64-bit access for SCX/SC and SCXE/SCE). The SC/SCE data address must be the address of the lowest byte in the double width memory access.

If the PC and PC+4 instruction encodings do not match, a Reserved Instruction exception is signaled. If the effective addresses of SCX and SC or SCXE and SCE are not 32-bit word aligned separately and 64-bit doubleword aligned together, then Address Error is signaled. See **Restrictions** section for a full description of match requirements, and special case for SDBBP and BREAK breakpoint instructions.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Restrictions:

The following restrictions apply to load-linked and store-conditional extended instructions in the LLX/SCX instruction family:

Coprocessor 0's *Cause* register bit *BD* is extended to indicate exceptions related to the next instruction after the LLX/SCX-family instruction. Pseudocode indicates what value *Cause.BD* should be set to via comments such as `SignalException(AddressError) /*BD=1*/`. Similarly, the status register *BadInstrP* is extended to hold the LLX/SCX-family instruction if an exception is signaled for the next instruction, with *BD*=1.

An LLX/SCX family instruction must be not be placed in a branch delay slot or compact branch forbidden slot: if this rule is violated, a Reserved Instruction exception will be signaled (with *EPC*=PC of branch, *BD*=1).

An LLX/SCX family instruction must be followed by a matching LL/SC-family instruction: An SCX instruction must be followed by an SC instruction of the same type. Similarly for LLX/LL, LLXE/LLE, and SCXE/SCE. If the following instruction does not match, a Reserved Instruction exception must be signaled (with *EPC*=PC of the LLX/SCX family instruction, *BD*=1).

Except: An LLX/SCX instruction may be followed by one of the breakpoint instructions BREAK or SDBBP, in which case the appropriate breakpoint exception takes priority over the Reserved Instruction exception. The BREAK exception will be signaled with *EPC*=PC of the LLX/SCX family instruction and *BD*=1. The debug exception caused by such an SDBBP will be reported with *DEPC*=PC of the LLX/SCX family instruction and *DBD*=1.

The *base* field must be the same in an LLX/SCX family instruction and the following, matching, LL/SC-family instruction: If the following instruction does not match, a Reserved Instruction exception must be signaled (with *EPC*=PC of the LLX/SCX family instruction, *BD*=1).

The *base* and *rt* fields of the LLX family instruction must not be the same. If they are the same a Reserved Instruction exception must be signaled (with *EPC*=PC of the LLX/SCX family instruction, *BD*=0).

The LLX/SCX and following LL/SC family instructions must match in their *offset* field: Given matching in instruction type and *base*, the difference between the *offset* fields of the instruction at PC and the instruction at PC+4 should be the data size, 4 for LLX/LLE/SCX/SCXE. Programmers should follow this rule in coding. However, implementations do not need to explicitly check this rule, since it is implied by other rules. TBD

Natural Alignment: The effective address must be naturally aligned for any LLX/SCX family instruction; if not naturally aligned, an Address Error exception is signaled. I.e. for LLX, LLXE, SCX and SCXE, if the two least significant bits of the effective address are not both zero, an Address Error exception is signaled. Such an Address Error exception is signaled with *EPC*=PC of the LLX/SCX family instruction, BD=0.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions such as LW (Load Word). However, this instruction is a special memory reference instruction for which misaligned support is NOT provided, and for which signalling an exception (AddressError) on a misaligned access is required.

Double Width Alignment: In addition to natural alignment, the memory bytes written by the LLX/SCX family instruction and the following LL/SC family instruction must be adjacent, non-overlapping, and must have the alignment natural for double the memory access size: The lowest byte address in an LLX/LL, LLXE/LLE, SCX/SC or SCXE/SCE pair must be 8-byte aligned. It is required that the LL/SC family instruction byte address be lower than that of the LLX/SCX family instruction. i.e. that the LL/SC family instruction in an LLX/LL or SCX/SC family instruction pair must be naturally aligned for double the memory access width.

The double width alignment condition must be satisfied for both virtual and physical addresses. If this condition is not met, then an Address Error exception is signaled, with EPC = PC of first instruction, and BD=1. This condition is guaranteed to be met in the physical address if met in the virtual address and if the SCX and SC translations are consistent.

Exception Priority: although LLX and LL may complete execution together, all exceptions for an LLX instruction (at PC) must be signaled, with *EPC*=PC and *BD*=0, before any exceptions are signaled, with *EPC*=PC and *BD*=1, for the

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

next instruction (at PC+4) or for any exceptions caused by the interaction between the LLX instruction and the next instruction. This is as if the LLX instruction is executed enough to signal all exceptions, followed by exception checks for the combination of LLX and the next instruction. Similarly for LLX/LL, LLXE/LLE, and SCXE/SCE instructions.

Exceptions relating to an LLX/SCX family instruction are reported with *EPC*=PC of the LLX/SCX family instruction, and *BD*=0.

Exceptions relating to interaction between an LLX/SCX family instruction and the following instruction are reported with *EPC*=PC of LLX/SCX instruction and *BD*=1.

Debug single step exceptions are reported with *DEPC*=PC of the LLX/SCX family instruction, and *BD*=0. No debug single step exception will be reported for the SC instruction of an SCX/SC pair: For the purposes of debug single stepping, the SCX/SC pair is atomic. Similarly for LLX/LL, LLE/LLXE, and SCXE/SCE pairs of instructions.

Exceptions related to the SCX/SC family instruction pair before following instruction cancel SCX but do ***not*** clear *LLbit*: if an exception or interrupt occurs at or after the SCX-family instruction and before or at the next instruction, the SCX is canceled, but *LLbit* is not cleared. I.e. the LLX/LL-SCX/SC atomic is not necessarily forced to fail. Exceptions are therefore reported with *EPC*=PC of SCX, and *BD*=0 or 1 as appropriate. Exception handling software should return (ERET or ERETNC) to the PC of the SCX instruction, re-executing the SCX/SC pair. Adjusting EPC or DEPC and returning to the SC instruction without re-executing the SCX instruction will result in incorrect behavior.

For exceptions related to an LLX/LL family instruction pair:

- No memory access is performed.
- Neither target register of the LLX/LL family instruction pair is updated.
- *LLbit* is not set.
- *EPC* (or *DEPC*) is set to the PC of the LLX family instruction.
- Status.BD is set to 0 or 1 as appropriate, as described below.

Exception handling software should return (ERET or ERETNC) to the PC of the LLX instruction, re-executing the LLX/LL pair. Adjusting EPC or DEPC and returning to the LL instruction without re-executing the LLX instruction will result in incorrect behavior.

LLX/LL and SCX/SC matching: the LL-family instruction, the SC-family instruction, and the optional LLX/SCX-family instructions in a MIPS atomic sequence *should*[1] match. Portable software should not rely on mismatching LLX/LL/SCX/SC to complete successfully, nor to fail. Implementations are permitted to cause the SC to fail if the LL/SCX/SC do not match, but are not required to do so. Matching LLX/LL/SCX/SC should be of the same instruction type (word (LLX/LL/SCX/SC), or word EVA (LLXE/LLE/SCXE/SCE)). Table 6.30 summarizes these rules for LL/SC family instructions.

1. Terminology: "*Should*" is a recommendation. Implementations are encouraged to provide *should* behavior, but are not required to do so. Portable software should not rely on such behavior, but is encouraged to follow *should* rules. "*Must*" behavior are requirements: Implementations are required to implement such behavior, and software that violates such requirements will fail, typically with a exception such as a Reserved Instruction exception or Address Error.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 6.30 Recommended and non-recommended LL/SC family instructions to start and end atomic code sequences

End of Atomic Sequence	Start of atomic sequence: LL	LLD	LLE	LLX /LL	LLDX /LLD	LLXE /LLE
SC	**OK**[1]	BAD	BAD	BAD	BAD	BAD
SCD	BAD[2]	**OK**	BAD	BAD	BAD	BAD
SCE	BAD	BAD	**OK**	BAD	BAD	BAD
SCX/SC	BAD	BAD	BAD	**OK**	BAD	BAD
SCDX/SCD	BAD	BAD	BAD	BAD	**OK**	BAD
SCXE/SCE	BAD	BAD	BAD	BAD	BAD	**OK**

1. Cells marked OK indicate recommended combinations of instructions to start and end LL/SC atomic code sequences.
2. Cells marked BAD (and shaded) indicate non-recommended combinations of instructions to start and end LL/SC atomic code sequences. Software should not be coded in this way. Implementations are not required to enforce this restriction, but software coded this way may succeed on some implementations, and fail on other implementations. I.e. success or failure of the SC family instruction is UNPREDICTABLE.

The LL and SC virtual and physical addresses should match completely. However, the memory addressing mode - the and offset - need not match between LLX/LL and SCX/SC. All physical address bits in the LL physical address and the corresponding bits in the SC physical address should match to the alignment required for the size of the LL/SC family instructions or LLX/LL and SCX/SC family instruction pairs.[2] This applies to atomic code sequences created via LL/SC, LLE/SCE, and their corresponding extended versions LLX/LL-SCX/SC, LLXE/LLE-SCXE/SC.

Translation Consistency: It is required that LL and SC match addresses, and that LLX/SCX family instructions lie in the same synchronization block. Even if all virtual addresses match, on a processor with hardware page table walking it is possible for physical address translation to change between LL and SC, and between the execution phase of LLX, LL, SCX and SC family instructions. e.g. between the time that SCX is first executed, and the time that the SCX store data is committed along with SC. The SCX/SC must only succeed if the SCX and SC physical addresses are consistent. If the address translations are inconsistent, implementations are required to fail the SCX/SC pair, or to retry them in a manner transparent to software. Similarly for LLX/LL pairs. Similarly for other information obtained from translation, such as the CCA (Cacheability and Coherence Attribute).

It is required that LLX/LL or SCX/SC instruction pairs act as if only a single address translation is done for the first instruction in the pair, and that translation is used for the second instruction, changing only lower address bits 3:0. Similarly for LLX/LL, LLXE/LLE, and SCXE/SCE instruction pairs.

Synchronizable memory type (CCA): 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.

2. Note that the implementation dependent *LLAddr* register (Load Linked Address (CP0 Register 17, Select 0)) does not hold physical address bits 0 to 4 as of Release 5 or after. The requirement all LL and SC address bits match therefore involves comparing LL address bits not stored in any software accessible register state.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

LLX/LL need not be writeable: The addressed location need not be writable for LL or LLX family instructions. If it is not writable a subsequent SC or SCX family instruction will fault, but LL or LLX family instructions may be used in situations that do not generate such faults, e.g. the PAUSE instruction.

LLX/LL and PAUSE: If an LLX/LL family instruction pair is followed by a PAUSE instruction, the PAUSE instruction must terminate if it cannot be guaranteed that any of the memory bytes address by the LLX/LL instruction pair have not been modified.

Memory Ordering of LL/SC family instructions (included LLX/SCX family instructions):

- An SCX/SC family instruction pair is executed atomically as seen by the processor executing these instructions and by other processors. I.e. the SC will not be seen to be executed before the SCX, and no other instruction, processor or device, can observe the SCX store without also being able to observe the SC store, or vice versa.
- LLX/LL family instruction pairs are not required to perform a double width atomic read of memory, but violations of atomicity will be detected, clearing LLbit, so that the matching SC will fail.[3]
  - Atomicity of LLX/LL family instruction pairs may be provided by MIPS CPU implementations as and if required by certain system configurations for uncached memory. [4]
- All LL/SC family instructions, including LLX/LL and SCX/SC family instruction pairs, are ordered by their implicit dependency on LLbit: e.g. a later LL will not be executed before an earlier SC from the same processor, even if their data memory addresses do not overlap.
- In the MIPS memory consistency architecture, LL/SC family instructions (including LLX/SCX family instructions) are not ordered with respect to other memory accesses from the same processor, except when their addresses overlap, or explicit SYNC instructions lie between them. E.g. a later LL can be executed before an earlier SW, or vice versa.[5]

Availability and Compatibility:

The LLX/SCX family of instructions is introduced by and required as of the MIPS Release 6 architecture and the microMIPS Release 6 architecture.

LLX and SCX are introduced by and required as of microMIPS32 Release 6. SCXE is introduced by and required as of microMIPS32 Release 6 when EVA is also implemented, which is indicated by bit *EVA* of coprocessor 0's *Config5* register.

3. For example, an implementation of LLX/LL in cached memory may have LLX set LLaddr and then perform the LLX word load, and then may execute LL separately. A separate processor may perform an atomic doubleword write that changes both the LLX and LL memory locations, such that the values returned by LLX and LL may not have both been simultaneously present in memory. However, if atomicity is violated in this way, then LLbit must be cleared. The LL instruction of an LLX/LL instruction pair will not set LLbit if it has been cleared after the LLX instruction. Overall, LLX/LL family instruction pairs are not required to be atomic; whereas SCX/SC family instruction pairs are required to be atomic, if performed.
   However, certain system configurations, for uncached memory in particular, require that the LLX/LL family instruction pair be performed atomically via a single bus transaction.
4. MIPS recommends that implementations perform a double width atomic read memory access for LLX/LL family instruction pairs, for cached as well as uncached memory, but does not require this. Portable software should not assume that an LLX/LL family instruction pair is atomic without using a matching SCX/SC family instruction pair to detect possible violations of atomicity.
5. Note that this applies also to ordinary load instructions lying between LL and SC, inside the atomic RMW sequence.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

The microMIPS Release 6 instruction encodings for the SCX family of instructions conflict with encodings used by valid instructions in microMIPS pre-Release 6: SCX conflicts with pre-Release 6 SWR, and SCXE conflicts with pre-Release 6 SWLE.

Operation:

```
/* pseudocode for SCX and for the following instruction;
 * this replaces the following instruction pseudocode.
 *
 * this_instruction = SCX instruction at PC during instruction time I
 * next_instruction = instruction at PC+4 during instruction time I
 *                  = instruction at PC during instruction time I+1
 *                  = SC, or BREAK or SDBBP, else invalid
 * 'SCX' and 'SC' are generic, applicable to SCX-family and SC-family.
 *
 * All exceptions are signaled with EPC or DEPC = PC of SCX instruction.
 * All exceptions in instruction time I are signaled with BD=0.
 * All exceptions in instruction time I+1 are signaled with BD=1.
 */
I: /* SCX-only execution in instruction time I */
   /* perform address calculation and translation and SCX-only checks. */
   successful_so_far ← 1

   if this_instruction is SCX then
      size ← 4
   else if this_instruction is SCXE then
      EVA_Checks() /*BD=0*/
      size ← 4
   else
      assert(IMPOSSIBLE)
   endif

   scx_va ← GPR[this_instruction.base] + sign_extend( this_instruction.offset )
   if scx_va & (size-1) ≠ 0 then SignalException(AddressError) /*BD=0*/ endif

   (scx_pa,scx_cca) ← AddressTranslation( scx_va, DATA, STORE ) /*BD=0*/

   scx_store_data ← GPR[this_instruction.rt]

   /* complete SCX execution in instruction time I+1 */

I+1:
   /* SCX execution time I+1 and next_instruction execution time I combined */
   /* All exceptions in instruction time I+1 are signaled with BD=1. */

   LLX_SCX_family_common_code(
          /*inputs:*/     this_instruction, scx_pa, scx_cca, size,
          /*returns:*/    next_instruction, sc_va, sc_pa, sc_cca
   )

   sc_store_data  ← GPR[next_instruction.rt]

   store_data_2xwide ← (scx_store_data << (size*8)) || sc_store_data

   /* Not shown: byte swapping default Little Endian to BigEndian, if needed */

   /* Required check that LL and SC physical addresses match (all bits) */
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
/* Note that LLAddr CP0 register may not hold full LL physical address */
if sc_pa_i ≠ LL physical address bit i for any bit i
   then successful_so_far ← 0 endif

/* Fundamental LLBit check for LL/SCX/SC */
if successful_so_far and LLbit = 1
then
   /* Optionally check that LL matches SCX/SC - opcode, size, etc. */
   StoreMemory( CCA, 2*size, store_data_2xwide, sc_pa, sc_va, DATA )
   scx_and_sc_successful ← 1
else
   scx_and_sc_successful ← 0
endif

GPR[next_instruction.rt] ← scx_and_sc_successful
LLbit ← 0
/* end of combined SCX / SC pseudocode */
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
where /* helper function */

function EVA_checks
   if (Config5EVA=0) then SignalException(ReservedInstruction) endif
   if !IsCoprocessorEnabled(0)
      then SignalException(CoprocessorUnusable, 0)endif
   AM = SegmentAM(address) /* TBD: bug in SCE pseudocode */
   if (AM != UUSK && AM != MUSK && AM != MUSUK)
      then SignalException(AddressError) endif
end function

function LLX_SCX_family_common_code (
   /*inputs: */ this_instruction, this_pa, this_cca, size,
   /*outputs:*/ next_instruction, next_va, next_pa, next_cca
)
   /* begin function */
   if next_instruction is BREAK or SDBBP then
      /* Execute BREAK or SDBBP in normal I+1 manner,
       * as if in a branch delay slot or compact branch forbidden slot.
       * signaling appropriate exception */
   endif

   /* next_instruction must be matching non-extended LL/SC family
    * - this pseudocode replaces normal pseudocode for next instruction. */
   if (this_instruction is LLX and next_instruction is not LL)
      or (this_instruction is LLXE and next_instruction is not LLE)
      or (this_instruction is SCX and next_instruction is not SC)
      or (this_instruction is SCXE and next_instruction is not SCE)
   then
      SignalException(ReservedInstruction) /*BD=1*/
endif
   /* next instruction is non-extended LL/SC family: consistency checks */

   /* Check base register field for consistency */
   if this_instruction.base ≠ next_instruction.base
      then SignalException(ReservedInstruction) /*BD=1*/ endif

   /* Address computation for LL/SC-family next_instruction */
   next_va ← GPR[next_instruction.base] + sign_extend( next_instruction.offset )

   /* LL/SC following LLX/SCX virtual address must be doublewidth aligned
   if next_va & (size*2-1) ≠ 0
      then SignalException(AddressError) /*BD=1*/ endif

   /* LLX/SCX and LL/SC address virtual addresses must be adjacent
    * (adjacent, nonoverlapping, doubleword aligned) */
   if this_va&(2*size-1) - next_va&(2*size-1) ≠size
      then SignalException(AddressError) /*BD=1*/ endif
   /* assert( this_va-next_va ≠ size ) */

   /* Check offsets for consistency */
   /* assert( this_instruction.offset - next_instruction.offset = size ) */
      /* offset check not needed - other constraints ensure */

   /* LL/SC virtual to physical address translation
   /* Reuse the translation of the first instruction to ensure consistency. */
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
    /* Note: after all RI and AE exceptions, for standard exception priority. */
    next_pa ← this_pa & (2*size-1)
       /* given alignment constraints,
        * next_pa = this_pa - size = this_pa & (2*size-1) */
    next_cca ← this_cca

end function /* LLX_SCX_family_common_code */
```

Exceptions:

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

Reserved Instruction

Programming Notes:

LL/SC (and LLX/SCX) code sequences function on multiprocessor systems for *cached coherent* memory.

LL/SC (and LLX/SCX) code sequences function on multiprocessor systems for *uncached* memory if the CPU supports bus transactions visible to external hardware so that such external hardware can guarantee that atomicity has not been violated. Such support is implementation dependent.

LL/SC (and LLX/SCX) code sequences 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, and so that violations of atomicity caused by exception handling can be detected.

LL/SC (and LLX/SCX) code sequences on a single processor for *uncached* memory so that parallel programs can be run on uniprocessor systems that do not support *cached* memory access types, and so that violations of atomicity caused by exception handling can be detected.

Example: MIPS32 64-bit compare and swap using LLX/LL-SCX/SC code sequence:

```
cas2x32_retry_loop:
   # (t0,t1) is value to be compared against value in memory at (tA,tA+4)
   # (t2,t3) is value to be written
   MOV    T2, T2'# add t2', r0, t2   # copy because SC destroys store data
   LLX    T5, (TA)4                  # load hi
   LL     T4, (TA)                   # load lo
   BNEC   T1, T5, cas2x32_fail       # compare hi
   NOP                               # CTI not allowed in forbidden slot
   BNEC   T0, T4, cas2x32_fail       # compare lo
   NOP                               # SCX not allowed in forbidden slot
   SCX    T3, (TA)4                  # store-conditional hi
   SC     T2', (TA)                  # store-conditional lo, checking for atomicity
   BEQZC  T2', cas2x32_retry_loop    # if not atomic (0), try again
cas2x32_fail:
```

Exceptions between the LLX/LL and SCX/SC may cause the 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. However, exceptions per se do not necessarily cause failure: the ERETNC instruction allows an exception handler to complete without clearing LLbit.

Example: MIPS32 64-bit atomic store using LLX/LL-SCX/SC code sequence:

```
# R1 = 64-bit aligned address, R2=lo 32 bits, R3=high 32 bits
st2x32_retry_loop:
   LLX    R5, (R1)4                  # throwing LLX/LL load data away
   LL     R5, (R1)
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
    MOV    R2, R2'                  # copy store data because SCX destroys
    SCX    R3, (R1)4                # store-conditional hi
    SC     R2', (R1)                # store-conditional lo, checking for atomicity
    BEQZC  R2', st2x32_retry_loop   # if not atomic (0), try again
# if we get here, then 64-bit store accomplished
```

MIPS recommends that the LLX/LL match the SCX/SC. Similarly LLXE/LLE should match SCXE/SCE. This recommendation may not be enforced by some implementations. If LLX/LL do not match LLDX/LLD, e.g. if there is only a single LL setting up for an SCX/SC pair, success or failure may be UNPREDICTABLE.

Example: MIPS32 64-bit atomic load using LLX/SCX:

```
# R1 = 64-bit aligned address, R2 and R3 will receive values loaded
ld2x32_retry_loop:
    LLX    R3, (R1)4
    LL     R2, (R1)
    MOV    R2, R2'
    SCX    R3, (R1)4                # store value read back
    SC     R2', (R1)                # store-conditional lo, checking for atomicity
    BEQZC  R4, ld2x32_retry_loop    # if not atomic (0), try again
# if we get here, then 64-bit load accomplished
```

Note that an SCX/SC instruction pair is required to test atomicity. Because atomicity cannot be tested without doing at least a SC store conditional instruction, this instruction sequence cannot be used to perform double width atomic reads from memory that the reader cannot write.

Example: MIPS32 64-bit atomic load using LL/SC without LLX/SCX:

```
# R1 = 64-bit aligned address, R2 and R3 will receive values loaded
ld2x32_retry_loop:
    LL     R2, (R12)
    SYNC
    LW     R3, (R13)
    MOV    R2, R2'
    SYNC
    SC     R2', (R12)# store-conditional lo, checking for atomicity
    BEQZC  R4,ld2x32_retry_loop # if not atomic (0), try again
# if we get here, then 64-bit load accomplished
```

Note that the load of (R2,R3) above is atomic in the sense that if the SC succeeds, then at some point between the LL and SC the values (R2,R3) were both present in memory at their corresponding memory locations (R12,R13). If (R12,R13) lie in the same synchronization block, then they are both present in memory at the time of the SC. If (R12,R13) are not in the same synchronization block, then while they were both present in memory at some time between LL and SC, the value of R13, the location which is not monitored by LL/SC, may have changed by the time of the SC.

Note also that SYNC instructions are needed between the LL and the LW, and between the LW and the SC, to prevent reordering of these memory accesses. Because such SYNCs are expensive, MIPS recommends the LLX/LL-SCX/SC code sequence over the LL-SYNC-LW-SYNC-SC code sequence.

Implementation Notes:

The synchronization block of memory used for LL/SC is typically the largest cache line in use.

Implementations of LL/SC in general, and LLX/LL-SCX/SC in particular, provide atomicity if the computer system can guarantee that, if the SC passes, then atomicity has not been violated by transactions between the LL and SC. It should also guarantee eventual success, i.e. that failures will not persist forever.

Correct implementation depends on the system, both the CPU and the external memory subsystem. For example, the CPU may implement LL/SC correctly for cacheable coherent memory, but if the I/O subsystem can write to memory

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

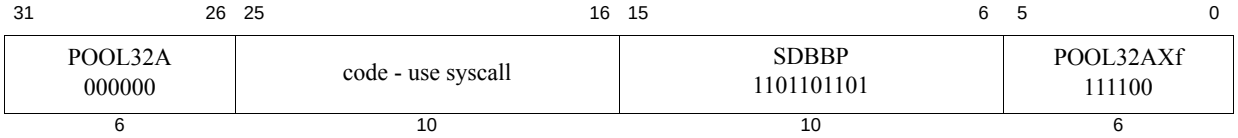
without being exposed to the cache coherency mechanism, LL/SC will not detect violations of atomicity caused by such non-coherent I/O accesses. Similarly, the CPU may implement uncached memory requests for LL and SC, but if the external memory subsystem performs an SC request and returns success without guaranteeing atomicity, LL/SC may not provide the expected guarantee of atomicity.

If it is not possible to guarantee such atomicity then it is recommended that implementations cause the SC to fail, returning the failure code in GPR[rt] without performing the store.

LL/SC and LLX/LL-SCX/SC code sequences should only be used for the following memory types (Cache and Coherency Attributes (CCAs)):

- *cached coherent*: if the cache protocol can guarantee that atomicity has not been violated by transactions between the LL and SC.
- *uncached:*
  - for uncached memory that is memory-like, i.e. which does not have memory-mapped I/O side effects
  - if the CPU supports bus transactions visible to external hardware so that such external hardware can guarantee that atomicity has not been violated by transactions between the LL and SC, and can signal success or failure by replying to the uncached bus transaction triggered by the SC-family instruction.
  - or if the system configuration is such that the CPU can observe all memory transactions that would violate atomicity
- *cached noncoherent* or *uncached* (no side effects): on uniprocessor systems lacking cache coherence or external hardware that can make atomicity assertions, LL-SC and LLX/LL-SCX/SC code sequences can be used to detect violations of atomicity caused by interrupt handling
- for other memory types: it may be **UNPREDICTABLE** whether the SC and possible SCX stores are performed, and whether the SC reports success or failure.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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:

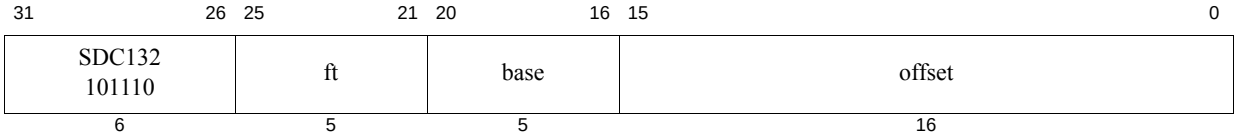
Operation:

```
NOP-ability for SDBBP. Also, signal hardware with ID
if Config5.SBRI=1 then /* SBRI is a MIPS Release 6 feature */
   SignalException(ReservedInstruction) endif
If Debug_DM = 1 then SignalDebugModeBreakpointException() endif // nested
SignalDebugBreakpointException() // normal
```

Exceptions:

Debug Breakpoint Exception
Debug Mode Breakpoint Exception

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SDC1 ft, offset(base)` **microMIPS**

Purpose: Store Doubleword from Floating Point

To store a doubleword from an FPR to memory.

Description: `memory[GPR[base] + offset] ← FPR[ft]`

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

Restrictions:

Pre-Release 6: An Address Error exception occurs if $EffectiveAddress_{2..0} \neq 0$ (not doubleword-aligned).

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

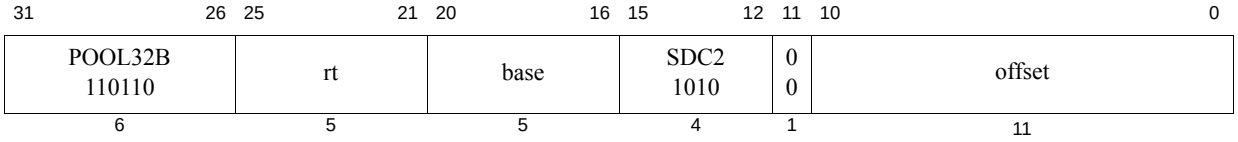
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_DOUBLEWORD)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SDC2 rt, offset(base)` **microMIPS**

Purpose: Store Doubleword from Coprocessor 2

To store a doubleword from a Coprocessor 2 register to memory

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

The 64-bit doubleword in Coprocessor 2 register *rt* is stored in memory at the location specified by the aligned effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

Pre-Release 6: An Address Error exception occurs if $EffectiveAddress_{2..0} \neq 0$ (not doubleword-aligned).

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
```

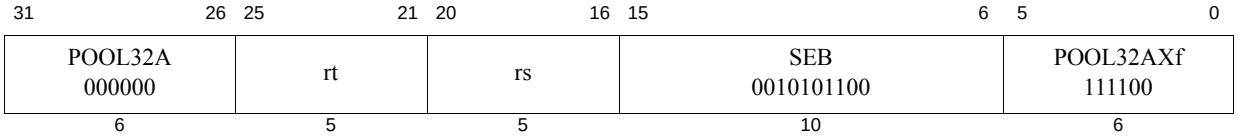
Exceptions:

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

Programming Notes:

As shown in the instruction drawing above, Release 6 implements an 11-bit offset, whereas all release levels lower than Release 6 of the MIPS architecture implement a 16-bit offset.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SEB rt, rs` **microMIPS**

Purpose: Sign-Extend Byte

To sign-extend the least significant byte of GPR *rs* and store the value into GPR *rt*.

Description: `GPR[rt] ← SignExtend(GPR[rs]7..0)`

The least significant byte from GPR *rs* is sign-extended and stored in GPR *rt*.

Restrictions:

Prior to architecture Release 2, this instruction resulted in a Reserved Instruction Exception.

Operation:

```
GPR[rt] ← sign_extend(GPR[rs]7..0)
```

Exceptions:

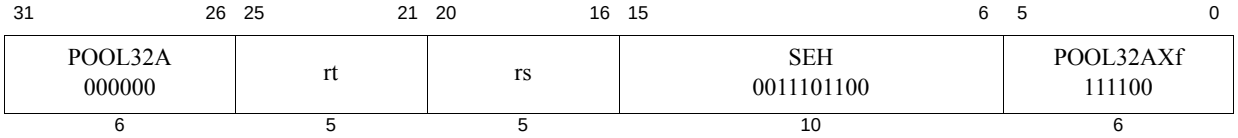
Reserved Instruction

Programming Notes:

For symmetry with the SEB and SEH instructions, you expect that there would be ZEB and ZEH instructions that zero-extend the source operand and expect that the SEW and ZEW instructions would exist to sign- or zero-extend a word to a doubleword. These instructions do not exist because there are functionally-equivalent instructions already in the instruction set. The following table shows the instructions providing the equivalent functions.

Expected Instruction	Function	Equivalent Instruction
`ZEB rx,ry`	Zero-Extend Byte	`ANDI rx,ry,0xFF`
`ZEH rx,ry`	Zero-Extend Halfword	`ANDI rx,ry,0xFFFF`

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SEH rt, rs` **microMIPS**

Purpose: Sign-Extend Halfword

To sign-extend the least significant halfword of GPR *rs* and store the value into GPR *rt*.

Description: `GPR[rt] ← SignExtend(GPR[rs]`$_{15..0}$`)`

The least significant halfword from GPR *rs* 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.

Operation:

```
GPR[rt] ← sign_extend(GPR[rs]15..0)
```

Exceptions:

Reserved Instruction

Programming Notes:

The SEH instruction can be used to convert two contiguous halfwords to sign-extended word values in three instructions. For example:

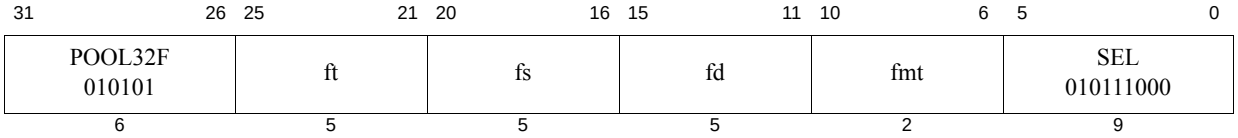
```
lw     t0, 0(a1)           /* Read two contiguous halfwords */
seh    t1, t0              /* t1 = lower halfword sign-extended to word */
sra    t0, t0, 16          /* t0 = upper halfword sign-extended to word */
```

Zero-extended halfwords can be created by changing the SEH and SRA instructions to ANDI and SRL instructions, respectively.

For symmetry with the SEB and SEH instructions, you expect that there would be ZEB and ZEH instructions that zero-extend the source operand and expect that the SEW and ZEW instructions would exist to sign- or zero-extend a word to a doubleword. These instructions do not exist because there are functionally-equivalent instructions already in the instruction set. The following table shows the instructions providing the equivalent functions.

Expected Instruction	Function	Equivalent Instruction
`ZEB rx,ry`	Zero-Extend Byte	`ANDI rx,ry,0xFF`
`ZEH rx,ry`	Zero-Extend Halfword	`ANDI rx,ry,0xFFFF`

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SEL.fmt`
`SEL fd, fs, ft, fmt` **microMIPS32 Release 6**

Purpose: Select floating point values with FPR condition

Description: `FPR[fd] ← FPR[fd].bit0 ? FPR[ft] : FPR[fs]`

SEL.fmt is a select operation, with a condition input in FPR `fd`, and 2 data inputs in FPRs `ft` and `fs`.

- If the condition is true, the value of `ft` is written to `fd`.
- If the condition is false, the value of `fs` is written to `fd`.

The condition input is specified by FPR `fd`, and is overwritten by the result.

The condition is true only if bit 0 of the condition input FPR `fd` is set. Other bits are ignored.

This instruction has floating point formats S and D, but these specify only the width of the operands. SEL.S can be used for 32-bit W data, and SEL.D can be used for 64 bit L data.

This instruction has no exception behavior. It does not trap on NaNs. It does not set the FPU Cause bits.

Restrictions:

None

Availability and Compatibility:

SEL.fmt is introduced by and required as of microMIPS32 Release 6.

Special Considerations:

Only formats S and D are valid. Other format values may be used to encode other instructions. Unused format encodings are required to signal the Reserved Instruction exception.

Operation:

```
tmp ← ValueFPR(fd, UNINTERPRETED_WORD)
cond ← tmp.bit0
if cond then
          tmp ← ValueFPR(ft, fmt)
else
          tmp ← ValueFPR(fs, fmt)
endif
StoreFPR(fd, fmt, tmp)
```

Exceptions:

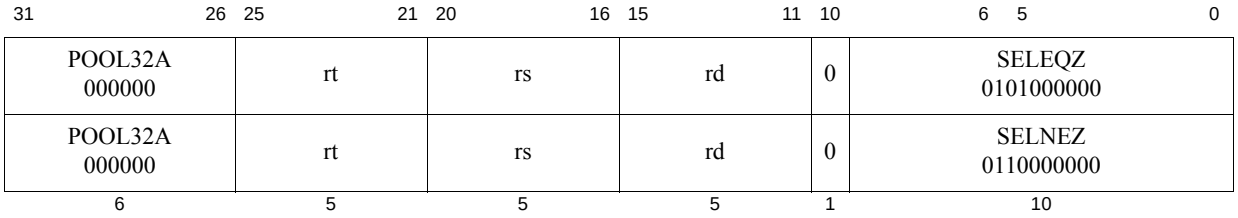
Coprocessor Unusable, Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Floating Point Exceptions:

Unimplemented Operation

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



31–26	25–21	20–16	15–11	10	6–5	0
POOL32A 000000	rt	rs	rd	0	SELEQZ 0101000000	
POOL32A 000000	rt	rs	rd	0	SELNEZ 0110000000	
6	5	5	5	1	10	

Format: `SELEQZ SELNEZ`

`SELEQZ rd,rs,rt` **microMIPS32 Release 6**

`SELNEZ rd,rs,rt` **microMIPS32 Release 6**

Purpose: Select integer GPR value or zero

Description:

```
SELEQZ: GPR[rd] ← GPR[rt]=0? GPR[rs]:0
SELNEZ: GPR[rd] ← GPR[rt] !=0?GPR[rs]:0
```

- SELEQZ is a select operation, with a condition input in GPR `rt`, one explicit data input in GPR `rs`, and implicit data input 0. The condition is true only if all bits in GPR `rt` are zero.
- SELNEZ is a select operation, with a condition input in GPR `rt`, one explicit data input in GPR `rs`, and implicit data input 0. The condition is true only if any bit in GPR `rt` is nonzero

If the condition is true, the value of `rs` is written to `rd`.

If the condition is false, the zero written to `rd`.

This instruction operates on all GPRLEN bits of the CPU registers, that is, all 32 bits on a 32-bit CPU, and all 64 bits on a 64-bit CPU. All GPRLEN bits of `rt` are tested.

Restrictions:

None

Availability and Compatibility:

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

Special Considerations:

None

Operation:

```
SELNEZ: cond ← GPR[rt] ≠ 0
SELEQZ: cond ← GPR[rt] = 0
if cond then
   result ←GPR[rs]
else
   result <= 0
endif
GPR[rd] ←result
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Programming Note:

Release 6 removes the Pre-Release 6 instructions MOVZ and MOVN:

```
MOVZ:  if GPR[rt] = 0 then GPR[rd] ← GPR[rs]
MOVN:  if GPR[rt] … 0 then GPR[rd] ← GPR[rs]
```

MOVZ can be emulated using Release 6 instructions as follows:

```
SELEQZ at, rs, rt
SELNEZ rd, rd, rt
OR rd, rd, at
```

Similarly MOVN:

```
SELNEZ at, rs, rt
SELEQZ rd, rd, rt
OR rd, rd, at
```

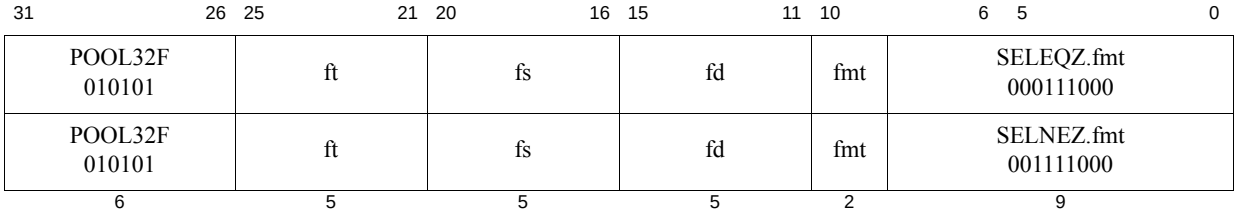
The more general select operation requires 4 registers (1 output + 3 inputs (1 condition + 2 data)) and can be expressed:

```
rD ← if rC then rA else rB
```

The more general select can be created using Release 6 instructions as follows:

```
    SELNEZ at, rB, rC
    SELNEZ rD, rA, rC
    OR rD, rD, at
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



31–26	25–21	20–16	15–11	10–6	5–0
POOL32F 010101	ft	fs	fd	fmt	SELEQZ.fmt 000111000
POOL32F 010101	ft	fs	fd	fmt	SELNEZ.fmt 001111000
6	5	5	5	2	9

Format: `SELEQZ.fmt SELNEQZ.fmt`

`SELEQZ.S fd,fs,ft` **microMIPS32 Release 6**

`SELEQZ.D fd,fs,ft` **microMIPS32 Release 6**

`SELNEZ.S fd,fs,ft` **microMIPS32 Release 6**

`SELNEZ.D fd,fs,ft` **microMIPS32 Release 6**

Purpose: Select floating point value or zero with FPR condition.

Description:

```
SELEQZ.fmt: FPR[fd] ← FPR[ft].bit0 ? 0 : FPR[fs]
SELNEZ.fmt: FPR[fd] ← FPR[ft].bit0 ? FPR[fs]: 0
```

- SELEQZ.fmt is a select operation, with a condition input in FPR `ft`, one explicit data input in FPR `fs`, and implicit data input 0. The condition is true only if bit 0 of FPR `ft` is zero.
- SELNEZ.fmt is a select operation, with a condition input in FPR `ft`, one explicit data input in FPR `fs`, and implicit data input 0. The condition is true only if bit 0 of FPR `ft` is nonzero.

If the condition is true, the value of `fs` is written to `fd`.

If the condition is false, the value that has all bits zero is written to `fd`.

This instruction has floating point formats S and D, but these specify only the width of the operands. Format S can be used for 32-bit W data, and format D can be used for 64 bit L data. The condition test is restricted to bit 0 of FPR `ft`. Other bits are ignored.

This instruction has no execution exception behavior. It does not trap on NaNs. It does not set the FPU Cause bits.

Restrictions:

FPR `fd` destination register bits beyond the format width are UNPREDICTABLE. For example, if `fmt` is S, then `fd` bits 0-31 are defined, but bits 32 and above are UNPREDICTABLE. For example, if `fmt` is D, then `fd` bits 0-63 are defined.

Availability and Compatibility:

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

Special Considerations:

Only formats S and D are valid. Other format values may be used to encode other instructions. Unused format encodings are required to signal the Reserved Instruction exception.

Operation:

```
tmp ← ValueFPR(ft, UNINTERPRETED_WORD)
SELEQZ: cond ← tmp.bit0 = 0
SELNEZ: cond ← tmp.bit0 ≠ 0
if cond then
          tmp ← ValueFPR(fs, fmt)
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

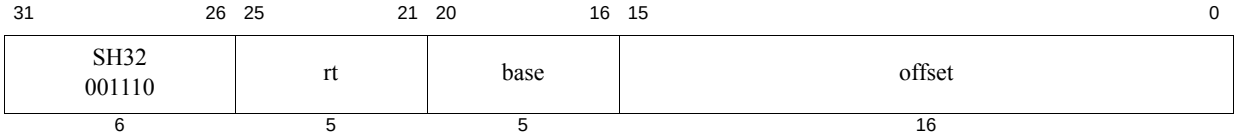
```
else
          tmp ← 0 /* all bits set to zero */
endif
StoreFPR(fd, fmt, tmp)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SH rt, offset(base)` **microMIPS**

Purpose: Store Halfword

To store a halfword to memory.

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

The least-significant 16-bit halfword of register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

Pre-release 6: The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

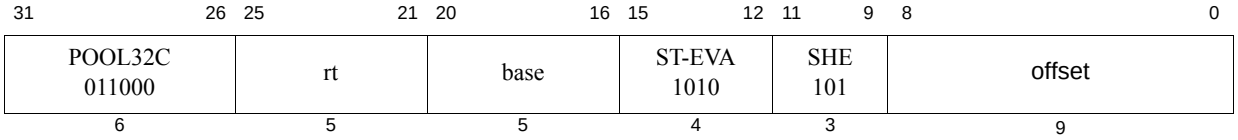
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1.. || (pAddr_..0 xor (ReverseEndian || 0))
bytesel ← vAddr_..0 xor (BigEndianCPU || 0)
dataword ← GPR[rt]_-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, HALFWORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

The least-significant 16-bit halfword of register *rt* is stored in memory at the location specified by the aligned effective address. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The SHE instruction functions the same 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 1.

Restrictions:

Only usable in kernel mode when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Pre-Release 6: The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

Operation:

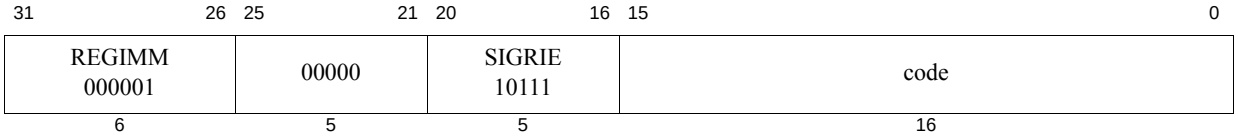
```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1..  || (pAddr_..0 xor (ReverseEndian || 0))
bytesel ← vAddr_..0 xor (BigEndianCPU || 0)
dataword ← GPR[rt]_-8*bytesel..0  || 0^8*bytesel
StoreMemory (CCA, HALFWORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Watch, Reserved Instruction, Coprocessor Unusable

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SIGRIE code` **MIPS32 Release 6**

Purpose: Signal Reserved Instruction Exception

The SIGRIE instruction signals a Reserved Instruction Exception.

Description: `SignalException(ReservedInstruction)`

The SIGRIE instruction signals a Reserved Instruction Exception. Implementations should use exactly the same mechanisms as they use for reserved instructions that are not defined by the Architecture.

The 16-bit *code* field is available for software use.

Restrictions:

The 16-bit *code* field is available for software use. The value zero in the *code* field should never be overloaded for any other purpose. Software may provide extended functionality by interpreting nonzero values of the *code* field in a manner that is outside the scope of this architecture specification.

Availability and Compatibility:

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

Pre-Release 6: this instruction encoding was reserved, and required to signal a Reserved Instruction exception. Therefore this instruction can be considered to be both backwards and forwards compatible.

The *rs* field of the SIGRIE instruction, bits 21-25,is required to be zero such as,`000001.00000.10011.code`.

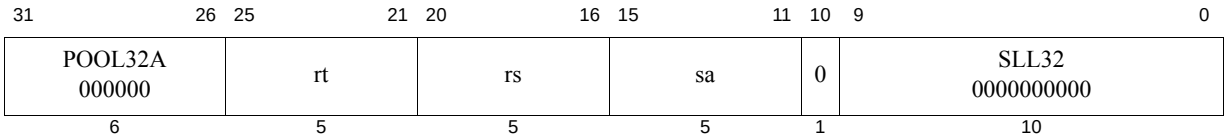
Operation:

```
SignalException(ReservedInstruction)
```

Exceptions:

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SLL rt, rs, sa` **microMIPS**

Purpose: Shift Word Left Logical

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

Description: `GPR[rt] ← GPR[rs] << sa`

The contents of the low-order 32-bit word of GPR *rs* are shifted left, inserting zeros into the emptied bits. The word result is placed in GPR *rt*. The bit-shift amount is specified by *sa*.

Restrictions:

None

Operation:

```
s ← sa
temp ← GPR[rs](31-s)..0 || 0^s
GPR[rt] ← temp
```

Exceptions:

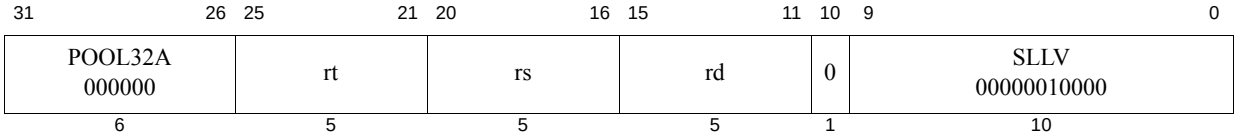
None

Programming Notes:

SLL r0, r0, 0, expressed as NOP, is the assembly idiom used to denote no operation.

SLL r0, r0, 1, expressed as SSNOP, is the assembly idiom used to denote no operation that causes an issue break on superscalar processors.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[rt] << GPR[rs]`

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeros into the emptied bits. The resulting word is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

None

Operation:

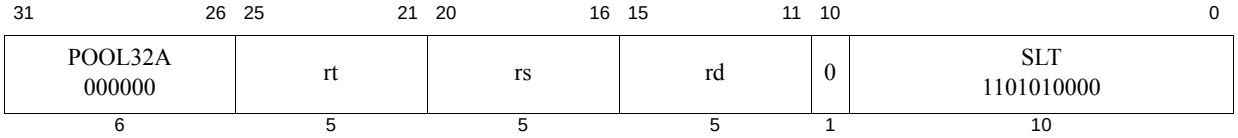
```
s ← GPR[rs]4..0
temp ← GPR[rt](31-s)..0 || 0s
GPR[rd] ← temp
```

Exceptions:

None

Programming Notes:

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SLT rd, rs, rt` **microMIPS**

Purpose: Set on Less Than

To record the result of a less-than comparison.

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

Compare the contents of GPR *rs* and GPR *rt* as signed integers; record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

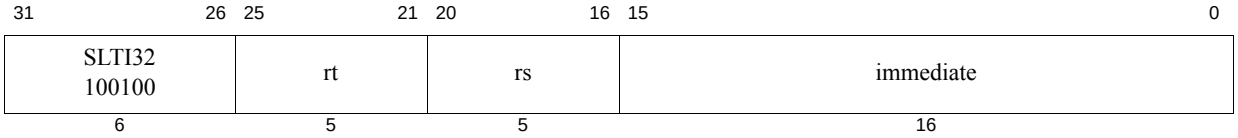
Operation:

```
if GPR[rs] < GPR[rt] then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← (GPR[rs] < immediate)`

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

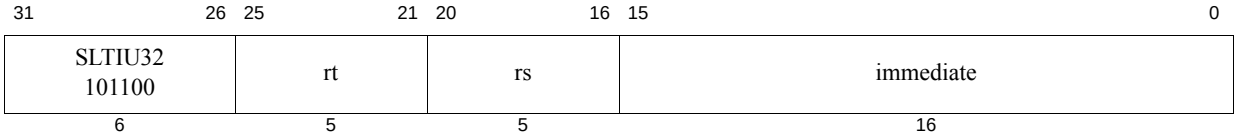
Operation:

```
if GPR[rs] < sign_extend(immediate) then
   GPR[rt] ← 0^(GPRLEN-1) || 1
else
   GPR[rt] ← 0^GPRLEN
endif
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SLTIU rt, rs, immediate` **microMIPS**

Purpose: Set on Less Than Immediate Unsigned

To record the result of an unsigned less-than comparison with a constant.

Description: `GPR[rt] ← (GPR[rs] < sign_extend(immediate))`

Compare the contents of GPR *rs* and the sign-extended 16-bit *immediate* as unsigned integers; record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

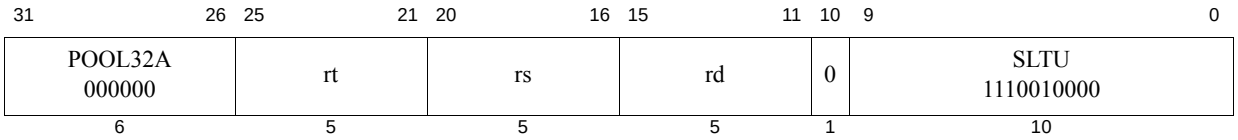
Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
   GPR[rt] ← 0^(GPRLEN-1) || 1
else
   GPR[rt] ← 0^GPRLEN
endif
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← (GPR[rs] < GPR[rt])`

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

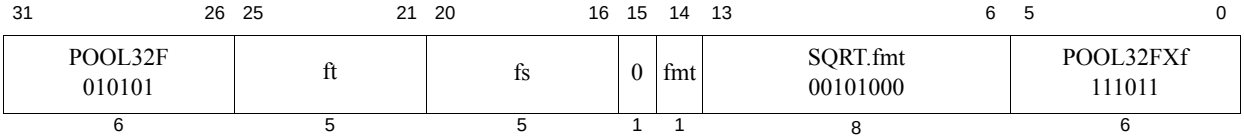
Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SQRT.fmt`

`SQRT.S ft, fs` **MIPS32**

`SQRT.D ft, fs` **MIPS32**

Purpose: Floating Point Square Root

To compute the square root of an FP value.

Description: `FPR[ft] ← SQRT(FPR[fs])`

The square root of the value in FPR *fs* is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *ft*. The operand and result are values in format *fmt*.

If the value in FPR *fs* corresponds to – 0, the result is – 0.

Restrictions:

If the value in FPR *fs* is less than 0, an Invalid Operation condition is raised.

The fields *fs* and *ft* must specify FPRs valid for operands of type *fmt*. If the fields 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, fmt, SquareRoot(ValueFPR(fs, fmt)))
```

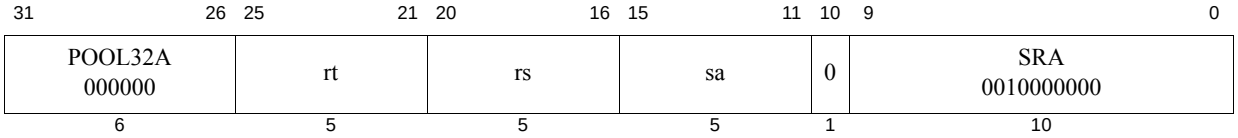
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Inexact, Unimplemented Operation

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[rs] >> 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 placed in GPR *rt*. The bit-shift amount is specified by *sa*.

Restrictions:

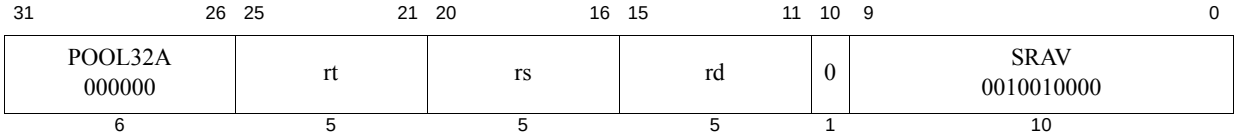
Operation:

```
s ← sa
temp ← GPR[rs]31)^s || GPR[rs]31..s
GPR[rt] ← temp
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[rt] >> GPR[rs]  (arithmetic)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

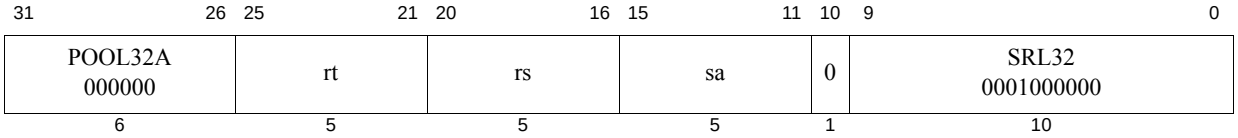
Operation:

```
s ← GPR[rs]4..0
temp ← (GPR[rt]31)s || GPR[rt]31..s
GPR[rd] ← temp
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[rs] >> 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 placed in GPR *rt*. The bit-shift amount is specified by *sa*.

Restrictions:

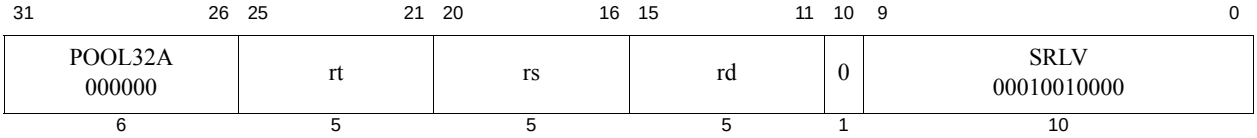
Operation:

```
s ← sa
temp ← 0^s || GPR[rs]_31..s
GPR[rt] ← temp
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← GPR[rt] >> GPR[rs]  (logical)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

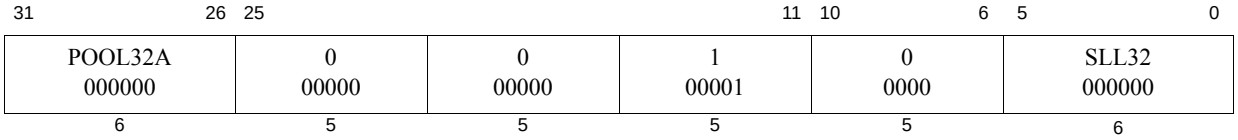
Operation:

```
s ← GPR[rs]4..0
temp ← 0^s || GPR[rt]31..s
GPR[rd] ← temp
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

Availability and Compatibility

Release 6: the special no-operation instruction SSNOP is deprecated: it behaves the same as a conventional NOP. Its special behavior with respect to instruction issue is no longer guaranteed. Release 6 requires interlocks, and the SYNC and JR.HB instructions are provided if stronger serialization are needed.

Assemblers targeting specifically Release 6 should reject the SSNOP instruction with an error.

Operation:

None

Exceptions:

None

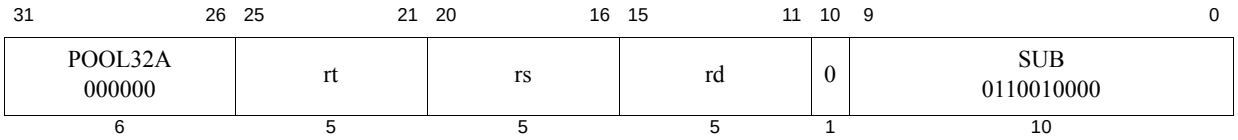
Programming Notes:

SSNOP is intended for use primarily to allow the programmer control over CP0 hazards by converting instructions into cycles in a superscalar processor. For example, to insert at least two cycles between an MTC0 and an ERET, one would use the following sequence:

```
mtc0   x,y
ssnop
ssnop
eret
```

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. 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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SUB rd, rs, rt` **microMIPS**

Purpose: Subtract Word

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

Description: `GPR[rd] ← GPR[rs] – GPR[rt]`

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* to produce a 32-bit result. If the subtraction results in 32-bit 2's complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

Operation:

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

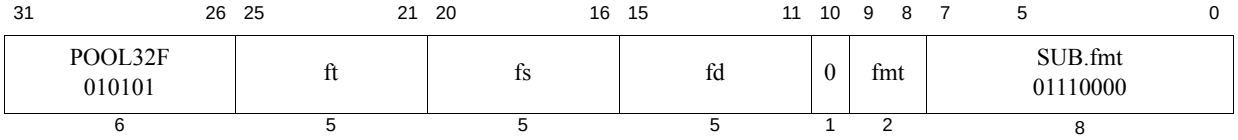
Exceptions:

Integer Overflow

Programming Notes:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SUB.fmt`

`SUB.S fd, fs, ft` **microMIPS**

`SUB.D fd, fs, ft` **microMIPS**

Purpose: Floating Point Subtract

To subtract FP values.

Description: `FPR[fd] ← FPR[fs] – FPR[ft]`

The value in FPR *ft* is subtracted from the value in FPR *fs*. The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. SUB.PS subtracts the upper and lower halves of FPR *fs* and FPR *ft* independently, and ORs together any generated exceptional conditions.

Restrictions:

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

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

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

Availability and Compatibility:

Not applicable.

Operation:

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

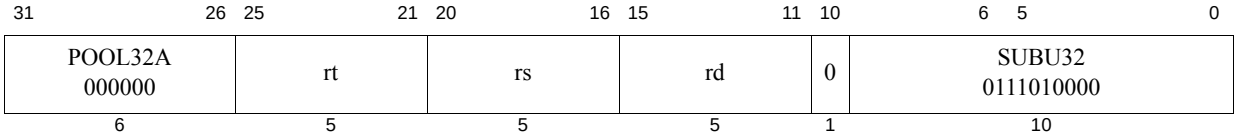
CPU Exceptions:

Coprocessor Unusable, Reserved Instruction

FPU Exceptions:

Inexact, Overflow, Underflow, Invalid Op, Unimplemented Op

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SUBU rd, rs, rt` **microMIPS**

Purpose: Subtract Unsigned Word

To subtract 32-bit integers.

Description: `GPR[rd] ← GPR[rs] – GPR[rt]`

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

No integer overflow exception occurs under any circumstances.

Restrictions:

Operation:

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

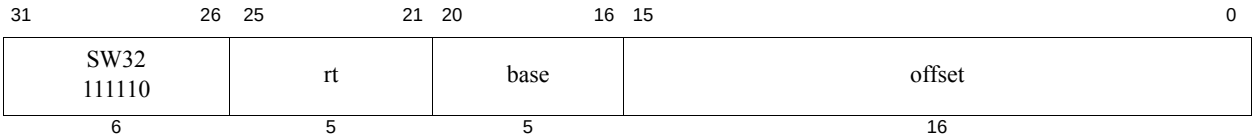
Exceptions:

None

Programming Notes:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SW rt, offset(base)` **microMIPS**

Purpose: Store Word

To store a word to memory.

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

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

Pre-Release 6: The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

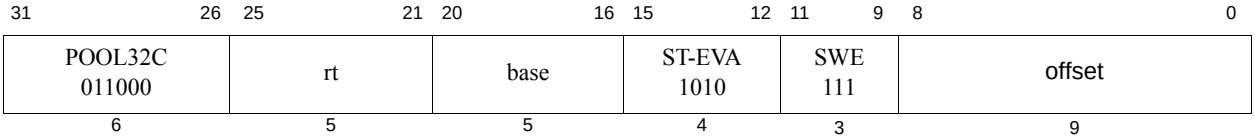
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 9-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

The SWE instruction functions the same 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 1.

Restrictions:

Only usable in kernel mode when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Pre-Release 6: The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

Operation:

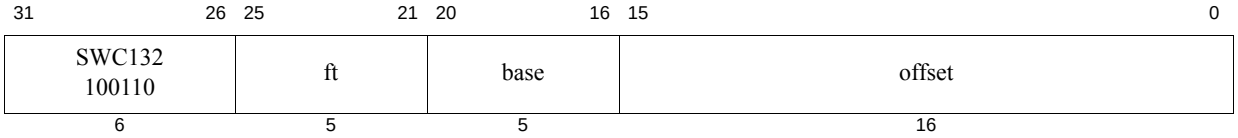
```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Watch, Reserved Instruction, Coprocessor Unusable

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.




```
SWC1 ft, offset(base)
```

microMIPS

Purpose: Store Word from Floating Point

To store a word from an FPR to memory.

Description: `memory[GPR[base] + offset] ← FPR[ft]`

The low 32-bit word from FPR *ft* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

Pre-Release 6: An Address Error exception occurs if $\text{EffectiveAddress}_{1..0} \neq 0$ (not word-aligned).

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

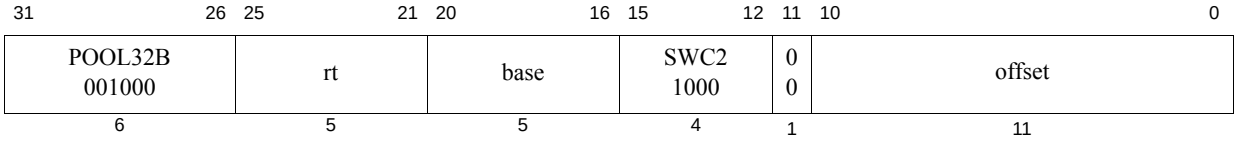
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
dataword ← ValueFPR(ft, UNINTERPRETED_WORD)
StoreMemory(CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SWC2 rt, offset(base)` **microMIPS**

Purpose: Store Word from Coprocessor 2

To store a word from a COP2 register to memory

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

The low 32-bit word from COP2 (Coprocessor 2) register *rt* is stored in memory at the location specified by the aligned effective address. The signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

Pre-Release 6: An Address Error exception occurs if $\text{EffectiveAddress}_{1..0} \neq 0$ (not word-aligned).

Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions. This instruction is considered an ordinary memory reference instruction for the purposes of misalignment: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

Note: The pseudocode in this instruction description may not yet have been updated to reflect misalignment support; the pseudocode may still indicate a required exception, which is now optional. See Appendix B, “Misaligned Memory Accesses” on page 511.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
dataword ← CPR[2,rt,0]
StoreMemory(CCA, WORD, dataword, pAddr, vAddr, DATA)
```

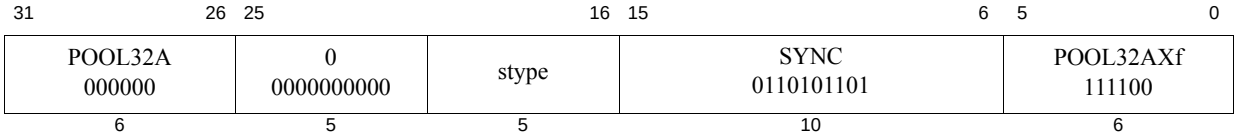
Exceptions:

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

Programming Notes:

As shown in the instruction drawing above, Release 6 implements an 11-bit offset, whereas all release levels lower than Release 6 of the MIPS architecture implement a 16-bit offset.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SYNC (stype = 0 implied)` **microMIPS**

`SYNC stype` **microMIPS**

Purpose: Synchronize Shared Memory

To order loads and stores for shared memory.

Description:

These types of ordering guarantees are available through the SYNC instruction:

- Completion Barriers
- Ordering Barriers

Completion Barrier — Simple Description:

- 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.

Completion Barrier — Detailed Description:

- Every synchronizable specified memory instruction (loads or stores or both) that occurs in the instruction stream before the SYNC instruction must be already globally performed before any synchronizable specified memory instructions that occur after the SYNC are allowed to be performed, with respect to any other processor or coherent I/O module.
- The barrier does not guarantee the order in which instruction fetches are performed.
- A stype value of zero will always be defined such that it performs the most complete set of synchronization operations that are defined.This means stype zero always does a completion barrier that affects both loads and stores preceding the SYNC instruction and both loads and stores that are subsequent to the SYNC instruction. Non-zero values of stype may be defined by the architecture or specific implementations to perform synchronization behaviors that are less complete than that of stype zero. If an implementation does not use one of these non-zero values to define a different synchronization behavior, then that non-zero value of stype must act the same as stype zero completion barrier. This allows software written for an implementation with a lighter-weight barrier to work on another implementation which only implements the stype zero completion barrier.
- A completion barrier is required, potentially in conjunction with SSNOP (in Release 1 of the Architecture) or EHB (in Release 2 of the Architecture), to guarantee that memory reference results are visible across operating mode changes. For example, a completion barrier is required on some implementations on entry to and exit from Debug Mode to guarantee that memory effects are handled correctly.

SYNC behavior when the stype field is zero:

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

- A completion barrier that affects preceding loads and stores and subsequent loads and stores.

Ordering Barrier — Simple Description:

- 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.

Ordering Barrier — Detailed Description:

- 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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 6.31 lists the available completion barrier and ordering barriers behaviors that can be specified using the stype field.

Table 6.31 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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

written by the store. The load by B must use the same memory access type as the store.

Globally performed load: A load instruction is *globally performed* when it is performed with respect to all processors and coherent I/O modules capable of storing to the location.

Globally performed store: A store instruction is *globally performed* when it is globally observable. It is *globally observable* when it is observable by all processors and I/O modules capable of loading from the location.

Coherent I/O module: A *coherent I/O module* is an Input/Output system component that performs coherent Direct Memory Access (DMA). It reads and writes memory independently as though it were a processor doing loads and stores to locations with a memory access type of *cached coherent*.

Load/Store Datapath: The portion of the processor which handles the load/store data requests coming from the processor pipeline and processes those requests within the cache and memory system hierarchy.

Restrictions:

The effect of SYNC on the global order of loads and stores for memory access types other than *uncached* and *cached coherent* is **UNPREDICTABLE**.

Operation:

```
SyncOperation(stype)
```

Exceptions:

None

Programming Notes:

A processor executing load and store instructions observes the order in which loads and stores using the same memory access type occur in the instruction stream; this is known as *program order*.

A *parallel program* has multiple instruction streams that can execute simultaneously on different processors. In multiprocessor (MP) systems, the order in which the effects of loads and stores are observed by other processors—the *global order* of the loads and store—determines the actions necessary to reliably share data in parallel programs.

When all processors observe the effects of loads and stores in program order, the system is *strongly ordered*. On such systems, parallel programs can reliably share data without explicit actions in the programs. For such a system, SYNC has the same effect as a NOP. Executing SYNC on such a system is not necessary, but neither is it an error.

If a multiprocessor system is not strongly ordered, the effects of load and store instructions executed by one processor may be observed out of program order by other processors. On such systems, parallel programs must take explicit actions to reliably share data. At critical points in the program, the effects of loads and stores from an instruction stream must occur in the same order for all processors. SYNC separates the loads and stores executed on the processor into two groups, and the effect of all loads and stores in one group is seen by all processors before the effect of any load or store in the subsequent group. In effect, SYNC causes the system to be strongly ordered for the executing processor at the instant that the SYNC is executed.

Many MIPS-based multiprocessor systems are strongly ordered or have a mode in which they operate as strongly ordered for at least one memory access type. The MIPS architecture also permits implementation of MP systems that are not strongly ordered; SYNC enables the reliable use of shared memory on such systems. A parallel program that does not use SYNC generally does not operate on a system that is not strongly ordered. However, a program that does use SYNC works on both types of systems. (System-specific documentation describes the actions needed to reliably share data in parallel programs for that system.)

The behavior of a load or store using one memory access type is **UNPREDICTABLE** if a load or store was previously made to the same physical location using a different memory access type. The presence of a SYNC between the references does not alter this behavior.

SYNC affects the order in which the effects of load and store instructions appear to all processors; it does not gener-

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

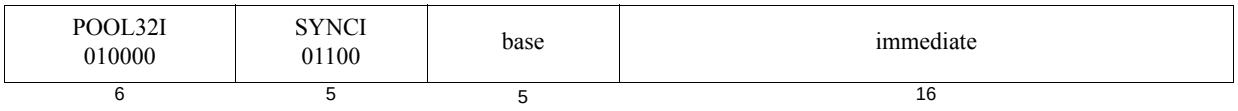
ally affect the physical memory-system ordering or synchronization issues that arise in system programming. The effect of SYNC on implementation-specific aspects of the cached memory system, such as writeback buffers, is not defined.

```
# 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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `SYNCI offset(base)` **microMIPS**

`SYNCI immediate (base)` **microMIPS Release 6**

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 by product 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 by product of this instruction. For example, if a writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error.

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

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

The operation of the processor is **UNPREDICTABLE** if the effective address references any instruction cache line that contains instructions to be executed between the SYNCI and the subsequent JALR.HB, JR.HB, or ERET instruction required to clear the instruction hazard.

The SYNCI instruction has no effect on cache lines that were 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.

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.

SYNCI globalization:

Release 6: SYNCI *globalization* (as described below) is required: compliant implementations must globalize SYNCI. Portable software can rely on this behavior, and use SYNCI rather than expensive “instruction cache shootdown” using inter-processor interrupts.

Pre-Release 6: portable software could not rely on the optional *globalization* (see below) of SYNCI meant that strictly portable software without implementation specific awareness could only rely on expensive “instruction cache shootdown using inter-processor interrupts.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

The SYNCI instruction acts on the current processor at a minimum. Implementations are required to affect caches outside the current processor to perform the operation on the current processor (as might be the case if multiple processors share an L2 or L3 cache).

In multiprocessor implementations where instruction caches are coherently maintained by hardware, the SYNCI instruction should behave as a NOP instruction.

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.

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:

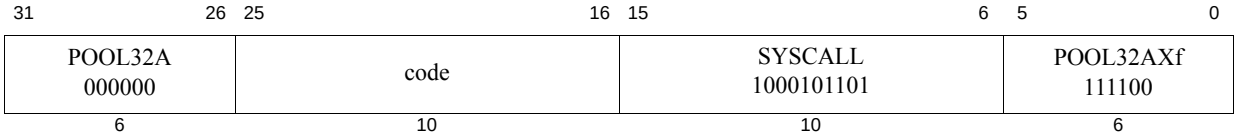
When the instruction stream is written, the SYNCI instruction should be used in conjunction with other instructions to make the newly-written instructions effective. The following example shows a routine which can be called after the new instruction stream is written to make those changes effective. 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 */
   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 */
   sltu   v1, a0, a1          /* Compare current with end address */
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
    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
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

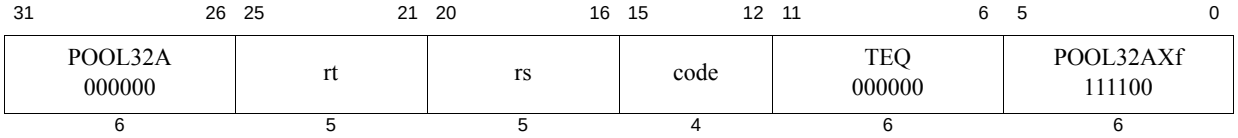
Operation:

```
SignalException(SystemCall)
```

Exceptions:

System Call

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

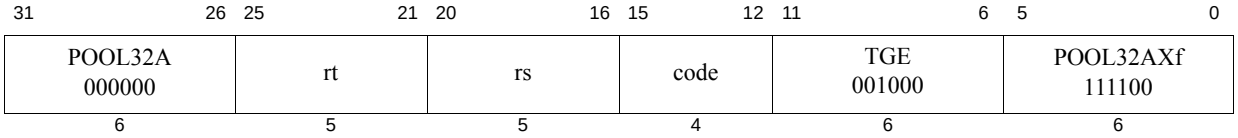
Operation:

```
if GPR[rs] = GPR[rt] then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `TGE rs, rt` **microMIPS**

Purpose: Trap if Greater or Equal

To compare GPRs and do a conditional trap.

Description: `if GPR[rs] ≥ GPR[rt] then Trap`

Compare the contents of GPR *rs* and GPR *rt* as signed integers. If GPR *rs* is greater than or equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, the system software must load the instruction word from memory.

Restrictions:

None

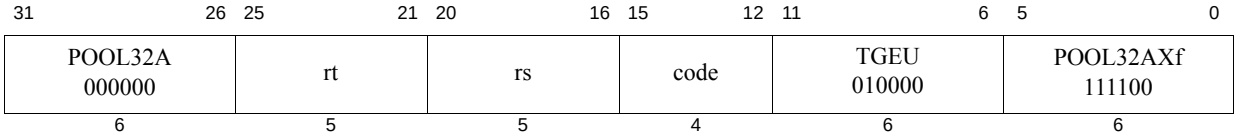
Operation:

```
if GPR[rs] ≥ GPR[rt] then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `TGEU rs, rt` **microMIPS**

Purpose: Trap if Greater or Equal Unsigned

To compare GPRs and do a conditional trap.

Description: `if GPR[rs] ≥ GPR[rt] then Trap`

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers. If GPR *rs* is greater than or equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, the system software must load the instruction word from memory.

Restrictions:

None

Operation:

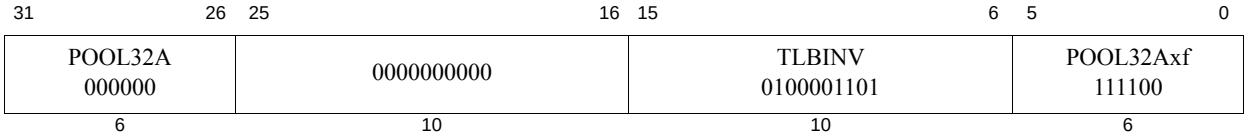
```
if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `TLBINV` **microMIPS**

Purpose: TLB Invalidate

TLBINV invalidates a set of TLB entries based on ASID and Index match. The virtual address is ignored in the entry match. TLB entries which have their G bit set to 1 are not modified.

Implementation of the TLBINV instruction is optional. The implementation of this instruction is indicated by the IE field in *Config4*.

Support for TLBINV is recommend for implementations supporting VTLB/FTLB type of MMU.

Implementation of $EntryHI_{EHINV}$ field is required for implementation of TLBGINV instruction.

Description:

On execution of the TLBINV instruction, the set of TLB entries with matching ASID are marked invalid, excluding those TLB entries which have their G bit set to 1.

The $EntryHI_{ASID}$ field has to be set to the appropriate ASID value before executing the TLBINV instruction.

Behavior of the TLBINV instruction applies to all applicable TLB entries and is unaffected by the setting of the *Wired* register.

- For JTLB-based MMU ($Config_{MT}$=1):

  All matching entries in the JTLB are invalidated. The *Index* register is unused.

- For VTLB/FTLB -based MMU ($Config_{MT}$=4):

  A TLBINV with the*Index* register set in VTLB range causes all matching entries in the VTLB to be invalidated.

  A TLBINV with the *Index* register set in FTLB range causes all matching entries in the single corresponding FTLB set to be invalidated.

  If TLB invalidate walk is implemented in software ($Config4_{IE}$=2), then software must do these steps to flush the entire MMU:

  1. one TLBINV instruction is executed with an index in VTLB range (invalidates all matching VTLB entries)
  2. a TLBINV instruction is executed for each FTLB set (invalidates all matching entries in FTLB set)

  If TLB invalidate walk is implemented in hardware ($Config4_{IE}$=3), then software must do these steps to flush the entire MMU:

  1. one TLBINV instruction is executed (invalidates all matching entries in both FTLB & VTLB). In this case, *Index* is unused.

Restrictions:

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of avail-

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

able TLB entries (For the case of *Config*$_{MT}$=4).

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

Availability and Compatibility:

Implementation of the TLBINV instruction is optional. The implementation of this instruction is indicated by the IE field in *Config4*.

Implementation of *EntryHI*$_{EHINV}$ field is required for implementation of TLBINV instruction.

Pre-Release 6, support for TLBINV is recommended for implementations supporting VTLB/FTLB type of MMU. Release 6 (and subsequent releases) support for TLBINV is required for implementations supporting VTLB/FTLB type of MMU.

Release 6: Processors that include a Block Address Translation (BAT) or Fixed Mapping (FM) MMU (Config$_{MT}$ = 2 or 3), the operation of this instruction causes a Reserved Instruction exception (RI).

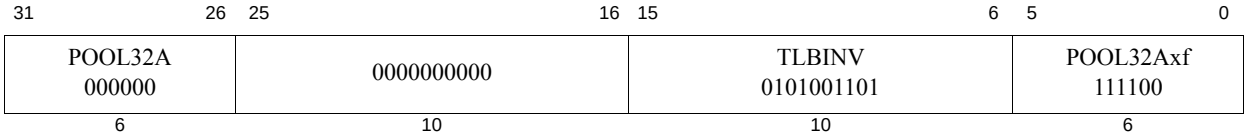
Operation:

```
if ( Config_MT=1 or (Config_MT=4 & Config4_IE=2 & Index ≤ Config1_MMU_SIZE-1))
   startnum ← 0
   endnum ← Config1_MMU_SIZE-1
endif
// treating VTLB and FTLB as one array
if (Config_MT=4 & Config4_IE=2 & Index > Config1_MMU_SIZE-1)
   startnum ← start of selected FTLB set // implementation specific
   endnum ← end of selected FTLB set - 1 //implementation specifc
endif

if (Config_MT=4 & Config4_IE=3))
   startnum ← 0
   endnum ← Config1_MMU_SIZE-1 + ((Config4_FTLBWays + 2) * Config4_FTLBSets)
endif

for (i = startnum to endnum)
   if (TLB[i].ASID = EntryHi_ASID & TLB[i].G = 0)
      TLB[i]_VPN2_invalid ← 1
   endif
endfor
```

Exceptions:

Coprocessor Unusable

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `TLBINVF` **microMIPS**

Purpose: TLB Invalidate Flush

TLBINVF invalidates a set of TLB entries based on *Index* match. The virtual address and ASID are ignored in the entry match.

Implementation of the TLBINVF instruction is optional. The implementation of this instruction is indicated by the IE field in *Config4*.

Support for TLBINVF is recommend for implementations supporting VTLB/FTLB type of MMU.

Implementation of the *EntryHIEHINV* field is required for implementation of TLBINV and TLBINVF instructions.

Description:

On execution of the TLBINVF instruction, all entries within range of *Index* are invalidated.

Behavior of the TLBINVF instruction applies to all applicable TLB entries and is unaffected by the setting of the *Wired* register.

- For JTLB-based MMU ($Config_{MT}$=1):

  TLBINVF causes all entries in the JTLB to be invalidated. *Index* is unused.

- For VTLB/FTLB-based MMU ($Config_{MT}$=4):

  TLBINVF with *Index* in VTLB range causes all entries in the VTLB to be invalidated.

  TLBINVF with *Index* in FTLB range causes all entries in the single corresponding set in the FTLB to be invalidated.

  If TLB invalidate walk is implemented in your software ($Config4_{IE}$=2), then your software must do these steps to flush the entire MMU:

  1. one TLBINV instruction is executed with an index in VTLB range (invalidates all VTLB entries)
  2. a TLBINV instruction is executed for each FTLB set (invalidates all entries in FTLB set)

  If TLB invalidate walk is implemented in hardware ($Config4_{IE}$=3), then software must do these steps to flush the entire MMU:

  1. one TLBINV instruction is executed (invalidates all entries in both FTLB & VTLB). In this case, *Index* is unused.

Restrictions:

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of available TLB entries ($Config4_{IE}$=2).

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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Availability and Compatibility:

Implementation of the TLBINV instruction is optional. The implementation of this instruction is indicated by the IE field in *Config4*.

Implementation of $EntryHI_{EHINV}$ field is required for implementation of TLBINV instruction.

Pre-Release 6, support for TLBINV is recommended for implementations supporting VTLB/FTLB type of MMU. Release 6 (and subsequent releases) support for TLBINV is required for implementations supporting VTLB/FTLB type of MMU.

Release 6: Processors that include a Block Address Translation (BAT) or Fixed Mapping (FM) MMU ($Config_{MT}$ = 2 or 3), the operation of this instruction causes a Reserved Instruction exception (RI).

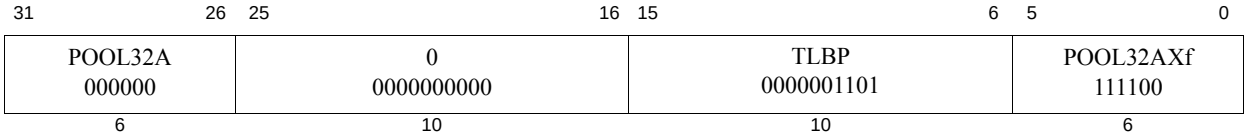
Operation:

```
if ( ConfigMT=1 or (ConfigMT=4 & Config4IE=2 & Index ≤ Config1MMU_SIZE-1))
   startnum ← 0
   endnum ← Config1MMU_SIZE-1
endif
// treating VTLB and FTLB as one array
if (ConfigMT=4 & Config4IE=2 & Index > Config1MMU_SIZE-1)
   startnum ← start of selected FTLB set // implementation specific
   endnum ← end of selected FTLB set - 1 //implementation specifc
endif

if (ConfigMT=4 & Config4IE=3))
   startnum ← 0
   endnum ← Config1MMU_SIZE-1 +((Config4FTLBWays + 2) * Config4FTLBSets)
endif

for (i = startnum to endnum)
   TLB[i]VPN2_invalid ← 1
endfor
```

Exceptions:

Coprocessor Unusable

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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.

Release 6: Processors that include a Block Address Translation (BAT) or Fixed Mapping (FM) MMU ($Config_{MT}$ = 2 or 3), the operation of this instruction causes a Reserved Instruction exception (RI).

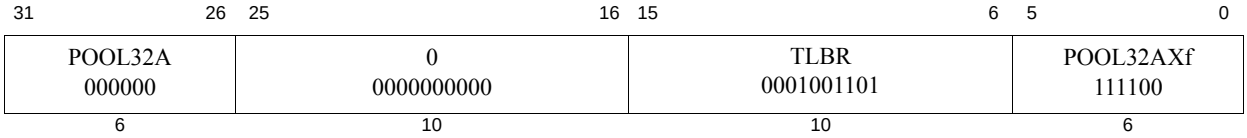
Operation:

```
Index ← 1 || UNPREDICTABLE31
for i in 0...TLBEntries-1
    if ((TLB[i]VPN2 and not (TLB[i]Mask)) =
                (EntryHiVPN2 and not (TLB[i]Mask))) and
        ((TLB[i]G = 1) or (TLB[i]ASID = EntryHiASID))then
        Index ← i
    endif
endfor
```

Exceptions:

Coprocessor Unusable, Machine Check

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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.

The value written to the *EntryHi*, *EntryLo0*, and *EntryLo1* registers may be different from the original written value 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.

Release 6: Processors that include a Block Address Translation (BAT) or Fixed Mapping (FM) MMU ($Config_{MT}$ = 2 or 3), the operation of this instruction causes a Reserved Instruction exception (RI).

Operation:

```
i ← Index
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

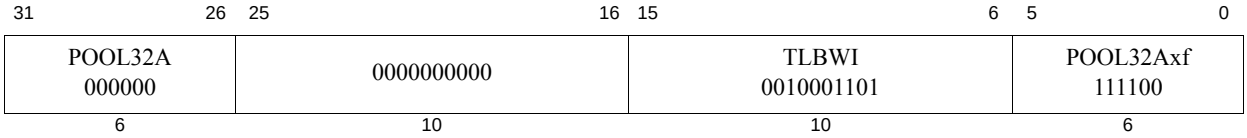
```
if i > (TLBEntries - 1) then
   UNDEFINED
endif
if ( (Config4IE = 2 or Config4IE = 3) and TLB[i]VPN2_invalid = 1) then
   PagemaskMask ← 0
   EntryHi ← 0
   EntryLo1 ← 0
   EntryLo0 ← 0
   EntryHiEHINV ← 1
else
   PageMaskMask ← TLB[i]Mask
   EntryHi ←
             (TLB[i]VPN2 and not TLB[i]Mask) || # Masking implem dependent
             05 || TLB[i]ASID
   EntryLo1 ← 0 ||
             (TLB[i]PFN1 and not TLB[i]Mask) || # Masking mplem dependent
             TLB[i]C1 || TLB[i]D1 || TLB[i]V1 || TLB[i]G
   EntryLo0 ← 0 ||
          (TLB[i]PFN0 and not TLB[i]Mask) || # Masking mplem dependent
          TLB[i]C0 || TLB[i]D0 || TLB[i]V0 || TLB[i]G
endif
```

Exceptions:

Coprocessor Unusable

Machine Check

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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.

Release 6: Processors that include a Block Address Translation (BAT) or Fixed Mapping (FM) MMU ($Config_{MT}$ = 2 or 3), the operation of this instruction causes a Reserved Instruction exception (RI).

Operation:

```
i ← Index
if (Config4IE = 2 or Config4IE = 3) then
   TLB[i]VPN2_invalid ← 0
   if ( EntryHIEHINV=1 ) then
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

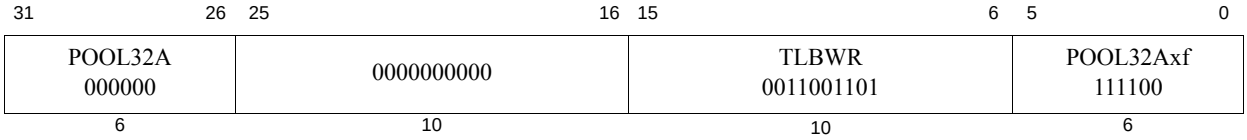
```
        TLB[i]VPN2_invalid ← 1
        break
    endif
endif
TLB[i]Mask ← PageMaskMask
TLB[i]VPN2 ← EntryHiVPN2 and not PageMaskMask # Implementation dependent
TLB[i]ASID ← EntryHiASID
TLB[i]G ← EntryLo1G and EntryLo0G
TLB[i]PFN1 ← EntryLo1PFN and not PageMaskMask # Implementation dependent
TLB[i]C1 ← EntryLo1C
TLB[i]D1 ← EntryLo1D
TLB[i]V1 ← EntryLo1V
TLB[i]PFN0 ← EntryLo0PFN and not PageMaskMask # Implementation dependent
TLB[i]C0 ← EntryLo0C
TLB[i]D0 ← EntryLo0D
TLB[i]V0 ← EntryLo0V
```

Exceptions:

Coprocessor Unusable

Machine Check

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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.

Release 6: Processors that include a Block Address Translation (BAT) or Fixed Mapping (FM) MMU ($Config_{MT}$ = 2 or 3), the operation of this instruction causes a Reserved Instruction exception (RI).

Operation:

```
i ← Random
if (Config4IE = 2 or Config4IE = 3) then
   TLB[i]VPN2_invalid ← 0
   endif
TLB[i]Mask ← PageMaskMask
TLB[i]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
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

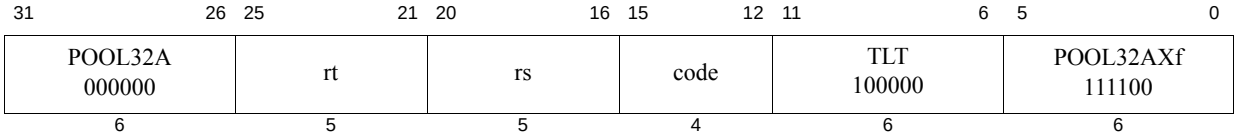
```
TLB[i]D0 ← EntryLo0D
TLB[i]V0 ← EntryLo0V
```

Exceptions:

Coprocessor Unusable

Machine Check

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

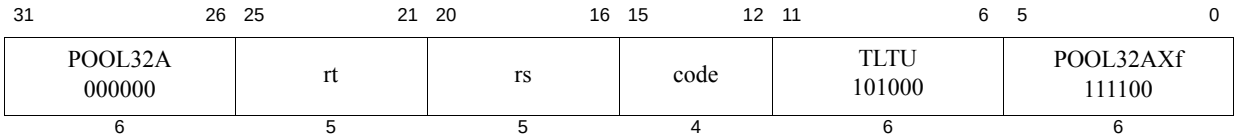
Operation:

```
if GPR[rs] < GPR[rt] then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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

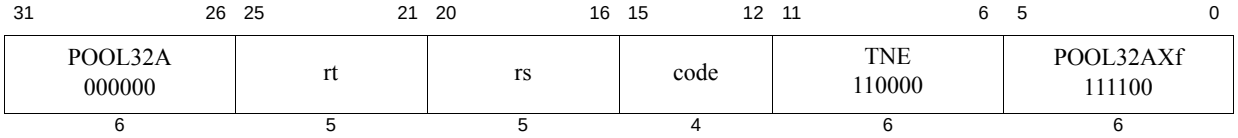
Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `TNE rs, rt` **microMIPS**

Purpose: Trap if Not Equal

To compare GPRs and do a conditional trap.

Description: `if GPR[rs] ≠ GPR[rt] then Trap`

Compare the contents of GPR *rs* and GPR *rt* as signed integers. If GPR *rs* is not equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

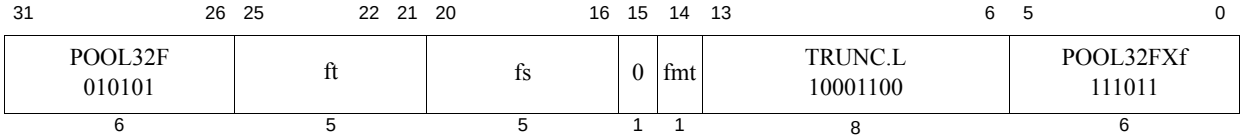
Operation:

```
if GPR[rs] ≠ GPR[rt] then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `TRUNC.L.fmt`

`TRUNC.L.S ft, fs` **microMIPS**

`TRUNC.L.D ft, fs` **microMIPS**

Purpose: Floating Point Truncate to Long Fixed Point

To convert an FP value to 64-bit fixed point, rounding toward zero.

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

The value in FPR *fs*, in format *fmt*, is converted to a value in 64-bit long-fixed point format and rounded toward zero (rounding mode 1). The result is placed in FPR *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 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, $2^{63}-1$, is written to *ft*.

Restrictions:

The fields *fs* and *ft* must specify valid FPRs: *fs* for type *fmt* and *fd* for long fixed point. If the fields 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))
```

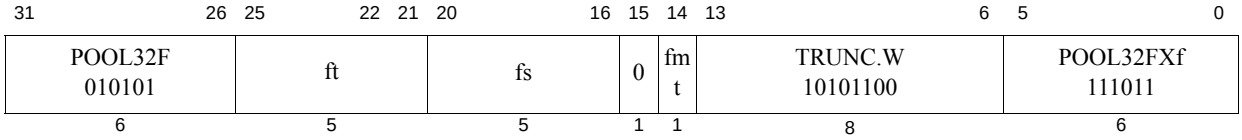
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `TRUNC.W.fmt`

`TRUNC.W.S ft, fs` **microMIPS**

`TRUNC.W.D ft, fs` **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] ← 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 the fields 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))
```

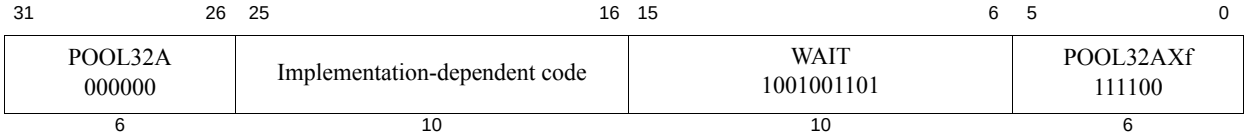
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Invalid Operation, Unimplemented Operation

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `WAIT` **microMIPS**

Purpose: Enter Standby Mode

Wait for Event

Description:

The WAIT instruction performs an implementation-dependent operation, involving a lower power mode. Software may use the code bits of the instruction to communicate additional information to the processor. 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 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:

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

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

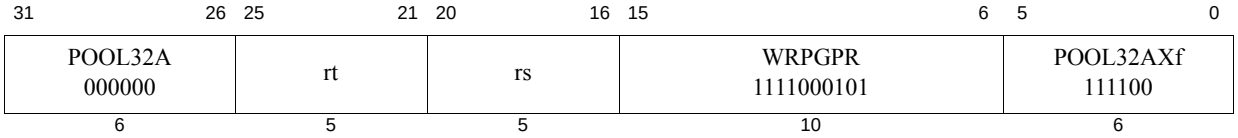
Operation:

```
I: Enter implementation dependent lower power mode
I+1:/* Potential interrupt taken here */
```

Exceptions:

Coprocessor Unusable Exception

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← 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:

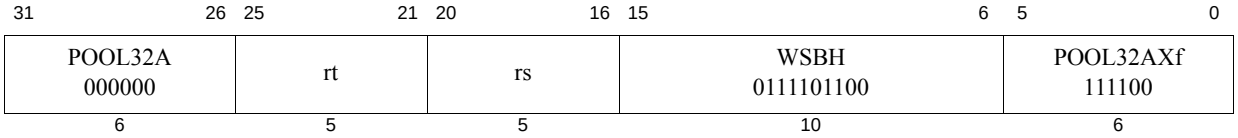
```
SGPR[SRSCtlPSS, rt] ← GPR[rs]
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



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] ← SwapBytesWithinHalfwords(GPR[rs])`

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

Restrictions:

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

Operation:

$GPR[rt] \leftarrow GPR[r]_{23..16} \,||\, GPR[r]_{31..24} \,||\, GPR[r]_{7..0} \,||\, GPR[r]_{15..8}$

Exceptions:

Reserved Instruction

Programming Notes:

The WSBH instruction can be used to convert halfword and word data of one endianness to another endianness. The endianness of a word value can be converted using the following sequence:

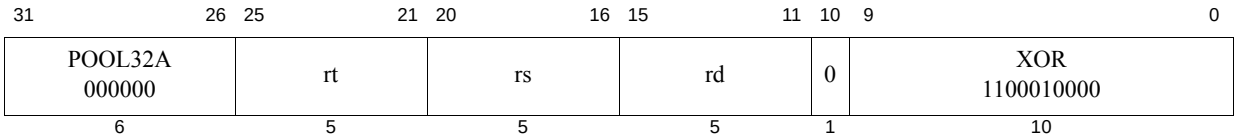
```
lw     t0, 0(a1)           /* Read word value */
wsbh   t0, t0              /* Convert endiannes of the halfwords */
rotr   t0, t0, 16          /* Swap the halfwords within the words */
```

Combined with SEH and SRA, two contiguous halfwords can be loaded from memory, have their endianness converted, and be sign-extended into two word values in four instructions. For example:

```
lw     t0, 0(a1)           /* Read two contiguous halfwords */
wsbh   t0, t0              /* Convert endiannes of the halfwords */
seh    t1, t0              /* t1 = lower halfword sign-extended to word */
sra    t0, t0, 16          /* t0 = upper halfword sign-extended to word */
```

Zero-extended words can be created by changing the SEH and SRA instructions to ANDI and SRL instructions, respectively.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `XOR rd, rs, rt` **microMIPS**

Purpose: Exclusive OR

To do a bitwise logical Exclusive OR.

Description: `GPR[rd] ← GPR[rs] XOR GPR[rt]`

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

Restrictions:

None

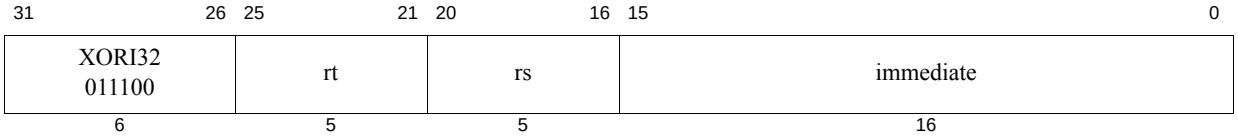
Operation:

```
GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.



Format: `XORI rt, rs, immediate` **microMIPS**

Purpose: Exclusive OR Immediate

To do a bitwise logical Exclusive OR with a constant.

Description: `GPR[rt] ← GPR[rs] XOR immediate`

Combine the contents of GPR *rs* and the 16-bit zero-extended *immediate* in a bitwise logical Exclusive OR operation and place the result into GPR *rt*.

Restrictions:

None

Operation:

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

Exceptions:

None

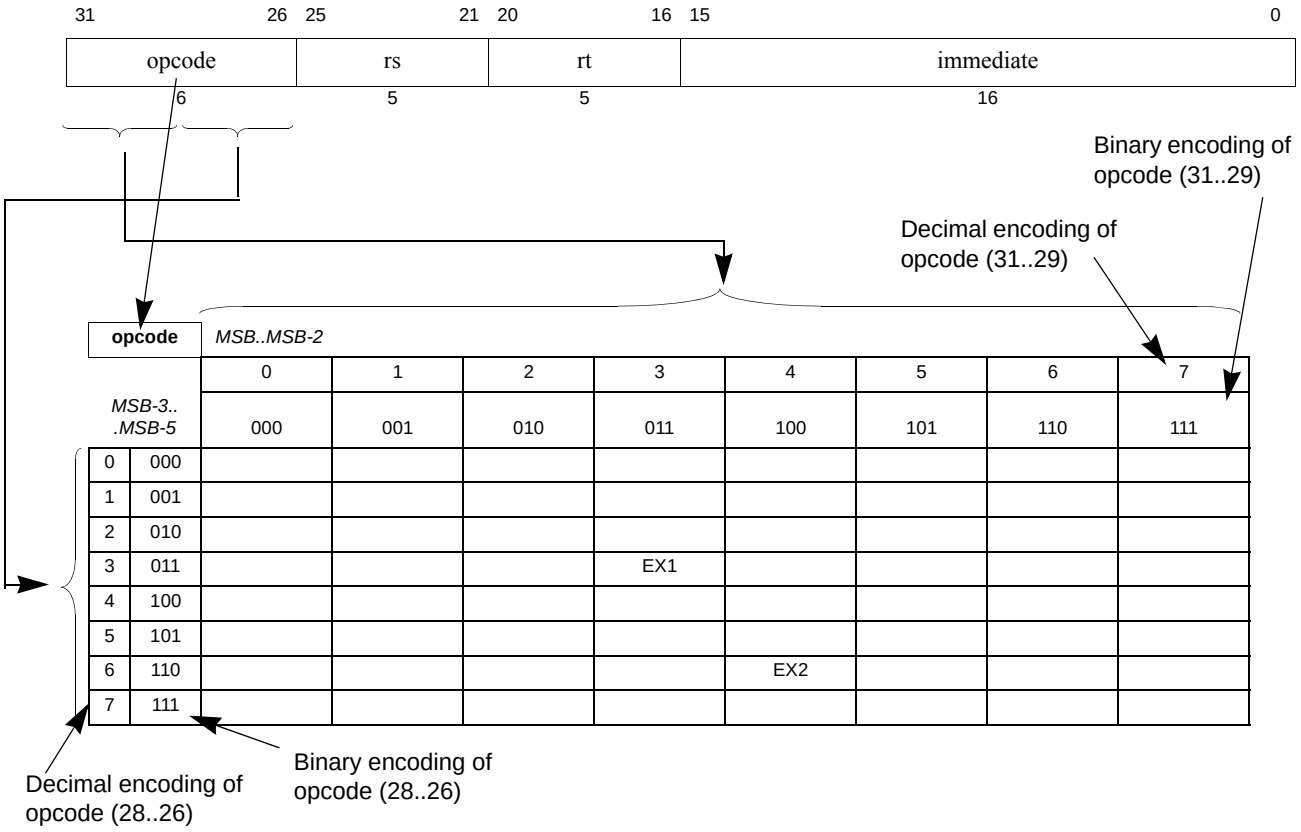
Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Opcode Map

This chapter defines the bit-level encoding of all microMIPS32 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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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.

Release 6 introduces additional nomenclature to the opcode tables for Release 6 instructions. For new instructions, bits 31:26 are generically named POP*XY* where *X* is the row number, and *Y* is the column number. This convention is extended to sub-opcode tables, except bits 5:0 are generically named SOP*XY* where *X* is the row number, and *Y* is the column number. This naming convention is applied where a specific encoded value are shared by multiple instructions.

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 *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
β	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level or a new revision of the Architecture. Executing such an instruction must cause a Reserved Instruction Exception.
∇	Operation or field codes marked with this symbol represent instructions which were only legal if 64-bit operations were enabled on implementations of Release 1 of the Architecture. In Release 2 of the architecture, operation or field codes marked with this symbol represent instructions which are legal if 64-bit floating point operations are enabled. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
Δ	Instructions formerly marked ∇ in some earlier versions of manuals, corrected and marked Δ in revision 5.03. Legal on MIPS64r1 but not MIPS32r1; in release 2 and above, legal in both MIPS64 and MIPS32, in particular even when running in “32-bit FPU Register File mode”, FR=0, as well as FR=1.
θ	Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, MIPS Technologies will assist the partner in selecting appropriate encodings if requested by the partner. The partner is not required to consult with MIPS Technologies when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (*SPECIAL2* encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
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.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 7.2 microMIPS32 Encoding of Major Opcode Field

Major	*MSB..MSB-2*	0	1	2	3	4	5	6	7
MSB-3.. MSB-5		000	001	010	011	100	101	110	111
0	000	POOL32A δ	POOL32B δ	POOL32i δ	POOL32C δ	BEQZC/JIC	BNEZC/JIALC	BLEZALC/ BGEZALC/ BGEUC	BGTZALC/ BLTZALC BLTUC
1	001	POOL16A δ	POOL16B δ	POOL16C δ	LWGP	POOL16F	*	*	*
2	010	LBU16	LHU16	LWSP16	LW16	SB16	SH16	SWSP16	SW16
3	011	MOVE16	ANDI16	POOL16D δ	POOL16E δ	BEQZC16	BNEZC16	BC16	LI16
4	100	AUI/LUI	ADDIU32	ORI32	XORI32	SLTI32	SLTIU32	ANDI32	β
5	101	LBU32	LHU32	POOL32F δ	BOVC/ BEQZALC/ BEQC	BC	BALC	BGTZC/ BLTZC/ BLTC	BLEZC/ BGEZC/ BGEC
6	110	SB32	SH32	POOL32S	ADDIUPC/ AUIPC ALUIPC/ LWPC	SWC132	β	β	SW32
7	111	LB32	LH32	β	BNVC/ BNEZALC/ BNEC	LWC132	β	β	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 7.13), 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 Instruction Set” on page 51.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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		*bit 5..3*									
		0	1	2	3	4	5	6	7		
bit 2..0		000	001	010	011	100	101	110	111		
										bit 9..6	
0	000	SLL32	*	SLLV	MUL	*	*	*	*	0000	0
0	000	SRL32	*	SRLV	MUH	*	*	*	*	0001	1
0	000	SRA	*	SRAV	MULU	*	*	*	*	0010	2
0	000	ROTR	*	ROTRV	MUHU	*	*	*	*	0011	3
0	000	*	*	ADD	DIV	*	*	*	*	0100	4
0	000	SELEQZ	*	ADDU32	MOD	*	*	*	*	0101	5
0	000	SELNEZ	*	SUB	DIVU	*	*	*	*	0110	6
0	000	*	*	SUBU32	MODU	*	*	*	*	0111	7
0	000	*	*	*	*	*	*	*	*	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	LSA	*	ALIGN	ε	*	*	*		

Not Shown

SLL r0, r0, r0 = NOP
SLL r0, r0, 1 = SSNOP
SLL r0, r0, 3 = EHB
SLL, r0, r0, 5 = PAUSE

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 7.5 POOL32Axf Encoding of Minor Opcode Extension Field

Extension	*bit 11..9*							
bit 8..6	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		

bit15..12

4	100	ε	ε	*	*	*	BITSWAP	*	JALRC	0000	0
4	100	ε	ε	*	*	*	*	*	JALRC.HB	0001	1
4	100	ε	*	*	*	*	SEB	*	*	0010	2
4	100	ε	*	*	*	*	SEH	*	*	0011	3
4	100	ε	*	*	*	*	CLO	MFC2	*	0100	4
4	100	ε	*	*	*	*	CLZ	MTC2	*	0101	5
4	100	ε	*	*	*	*	RDHWR		*	0110	6
4	100	ε	ε	*	*	*	WSBH		*	0111	7
4	100		*	*	*	*	*	MFHC2	*	1000	8
4	100	ε	ε	*	*	*	*	MTHC2	*	1001	9
4	100		*	*	*	*	*	*	*	1010	a
4	100	ε	ε	*	*	*	*	*	*	1011	b
4	100	*	*	*	*	*	*	CFC2	*	1100	c
4	100	ε	ε	*	*	*	*	CTC2	*	1101	d
4	100	*	*	*	*	*	*	*	*	1110	e
4	100	ε	*	*	*	*	*	*	*	1111	f

bit15..12

5	101	*	TLBP	ε	*	*	*	*	*	0000	0
5	101	*	TLBR	ε	*	*	*	*	*	0001	1
5	101	*	TLBWI	ε	*	*	*	*	*	0010	2
5	101	*	TLBWR	ε	*	*	*	*	*	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

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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	RINT.fmt	*	ADD.fmt	SELEQZ.fmt	000	0
0	000	*	*	*	ε	CLASS.fmt	*	SUB.fmt	SELNEZ.fmt	001	1
0	000	*	*	*	ε	*	*	MUL.fmt	SEL.fmt	010	2
0	000	*	*	*	ε	*	*	DIV.fmt	*	011	3
0	000	*	*	*		*	*	ADD.fmt	*	100	4
0	000	*	*	*		*	*	SUB.fmt	*	101	5
0	000	CVT.PS.S ∇	*	*	*	*	*	MUL.fmt	MADDF.fmt	110	6
0	000	*	*	*	*	*	*	DIV.fmt	MSUBF.fmt	111	7

1	001	*	*	*	*	*	*	*	*

2	010	*	*	*	*	*	*	*	*

3	011	MIN.fmt	MAX.fmt	*	*	*	MAXA.fmt	*	POOL32Fxf δ

4	100	*	*	*	ε	*	*	*	*

5	101	COMP.cond.S	*	CMP.cond.D	*	*	*	*	*

6	110	*	*	*	*	*	*	*	*

7	110	*	*	*	*	*	*	*	*

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 7.7 POOL32Fxf Encoding of Minor Opcode Extension Field

Extension		bit10..8									
		0	1	2	3	4	5	6	7		
bit 7..6		000	001	010	011	100	101	110	111		
										bit 13..11	
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	*	*	TRUNC.L.fmt ∇		*		*	100	4
0	00	MTC1	*	*	TRUNC.W.fmt		*	*	*	101	5
0	00	MFHC1 ∇	*	*	ROUND.L.fmt ∇		*	*	*	110	6
0	00	MTHC1 ∇	*	*	ROUND.W.fmt	*	*	*	*	111	7
										bit 12..11	
1	01	MOV.fmt	MOVF	*	ABS.fmt	*	*	*	ε	00	0
1	01	*	MOVT	*	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		bit 15	
		0	1
bit 14.12		0	1
0	000	LWC2	SWC2
1	001	LWP	SWP
2	010		
3	011	ε	ε
4	100		
5	101	LWM32	SWM32
6	110	CACHE	*
7	111		

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 7.9 POOL32C Encoding of Minor Opcode Field

Minor		bit 15	
bit 14..12		0	1
0	000	*	*
1	001	LLX	SCX
2	010	PREF	ST-EVA δ
3	011	LL	SC
4	100	*	*
5	101	β	β
6	110	β	β
7	111	β	β

Table 7.10 LD-EVA Encoding of Minor Opcode Field

Minor		
bit 11..9		
0	000	LBUE
1	001	LHUE
2	010	LLXE
3	011	*
4	100	LBE
5	101	LHE
6	110	LLE
7	111	LWE

Table 7.11 ST-EVA Encoding of Minor Opcode Field

Minor		
bit 11..9		
0	000	SCXE
1	001	*
2	010	PREFE
3	011	CACHEE
4	100	SBE
5	101	SHE
6	110	SCE
7	111	SWE

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 7.12 POOL32I Encoding of Minor Opcode Field

Minor		bit 22..21			
		0	1	2	3
bit 25..23		00	01	10	11
0	000	BNZ.df	BNZ.df	BNZ.df	BNZ.df
1	001	BZ.df	BZ.df	BZ.df	BZ.df
2	010	BC1EQZ	BC1NEZC	BC2EQZC	BC2NEZC
3	011	SYNCI	*	ε	ε
4	100	*	*	BNZ.V	BZ.V
5	101	*	*	*	*
6	110	*	*	ε	ε

						bit16
7	111	*	*	*	*	0
7	111	ε	ε	ε	ε	1

Table 7.13 POOL16A Encoding of Minor Opcode Field

Minor	
bit 0	
0	ADDU16
1	SUBU16

Table 7.14 POOL16B Encoding of Minor Opcode Field

Minor	
bit 0	
0	SLL16
1	SRL16

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 7.15 POOL16C Encoding of Minor Opcode Field

Minor		bit 6..4							
		0	1	2	3	4	5	6	7
bit 9..7		000	001	010	011	100	101	110	111
0	000	NOT16	XOR16	NOT16	XOR16	NOT16	XOR16	NOT16	XOR16
1	001	AND16	OR16	AND16	OR16	AND16	OR15	AND16	OR16
2	010	LWM16	SWM16	LWM16	SWM16	LWM16	SWM16	LWM16	SWM16
3	011	JRC16	JALRC16	JRCADDIUSP	BREAK16	JRC16	JALRC16	JRCADDIUSP	SDBBP16
4	100	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP
5	101	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP
6	110	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP
7	111	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP	MOVEP

For Release 6:

- NOT16, AND16, XOR16, OR16, LWM16, SWM16, and BREAK16, and SDBBP16 instructions have been repositioned in POOL16C and are not shown in the above table.
- JRADDIUSP has been converted to JRCADDIUSP and repositioned in POOL16C.
- MOVEP has moved from POOL16F to POOL16C.

Table 7.16 POOL16D Encoding of Minor Opcode Field

Minor	
bit 0	
0	ADDIUS5
1	ADDIUSP

Table 7.17 POOL16E Encoding of Minor Opcode Field

Minor	
bit 0	
0	ADDIUR2
1	ADDIUR1SP

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 7.18 POOL16F Encoding of Minor Opcode Field

Minor	
bit 0	
0	*
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 7.19 Floating Point Unit Format Encodings - S, D, PS

fmt 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 7.20 Floating Point Unit Format Encodings - S, D 1-bit

fmt field		Mnemonic	Name	Bit Width	Data Type
Decimal	Hex				
0	0	S	Single	32	Floating Point
1	1	D	Double	64	Floating Point

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Table 7.21 Floating Point Unit Instruction Format Encodings - S, D 2-bits

fmt 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 7.22 Floating Point Unit Format Encodings - S, W, L

fmt 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 7.23 Floating Point Unit Format Encodings - D, W, L

fmt 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.			

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Chapter 8

Compatibility

This chapter covers various aspects of compatibility. microMIPS is the preferred replacement for the existing MIPS16e ASE and uses the same mode-switch mechanism. Although microMIPS includes almost all MIPS instructions and therefore does not require the original MIPS encodings, initially it will be implemented together with MIPS-encoded instruction execution.

8.1 Assembly-Level Compatibility

microMIPS includes a re-encoding of the MIPS instructions, including all ASEs and UDI space. Therefore, microMIPS provides assembly-level compatibility. Only the following cases cause some side effects:

- **Re-encoded MIPS 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 MIPSencoding.

- **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 MIPS-encoded instructions, and because all branch target addresses of the MIPS 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.

- **MIPS assembly instructions manually encoded using the .WORD directive**

  Manual encoding of MIPS 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:

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

```
.set micromips   ; instruction stream is microMIPS

.set nomicromips ; instruction stream is MIPS

.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 MIPS NOP:

```
    .set nomicromips
    .word 0      ; represents MIPS NOP instruction
```

For MIPS 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 MIPS encoding as well as MIPS16e. This guarantees that static and dynamic linking processes can link microMIPS object files with MIPS object files.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Programs can be composed of both microMIPS and MIPS 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 2.5, "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. MIPS 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 MIPS) 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 microMIPS and MIPS

In some systems, it would be advantageous to place both microMIPS and MIPSexecutables 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 MIPSinstruction set, this binary sequence is interpreted as:

```
BEQ $0, $0, wxyz // branch to location of more MIPSinstructions
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
```


Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

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 2.6, "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 MIPS requires instructions to be 32-bit aligned, there is no forward compatibility issue when transitioning to microMIPS.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Appendix A

References

This appendix lists other publications available from Imagination 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.imgtec.com.

- MIPS® Architecture For Programmers, Volume I: Introduction to the MIPS32® Architecture
- MIPS® Architecture For Programmers, Volume II: The MIPS32® Instruction Set
- MIPS® Architecture For Programmers, Volume III: The MIPS32® and microMIPS32™ Privileged Resource Architecture

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Appendix B

Revision History

Revision	Date	Description
1.08	November 25, 2009	• Clean-up for external release.
1.09	January 7, 2010	• Added shared boot-up code sequence in Compatibility Chapter.
3.00	March 25, 2010	• 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	• 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	• POOL32Sxf binary encoding was incorrect for individual instruction description pages.
3.03	December 10, 2010	• microMIPS AFP versions security reclassification.
3.04	March 21, 2011	• 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	• 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	• CVT.D.fmt and CVT.S.fmt were in wrong positions within Table POOL32Fxf.
3.07	October 26, 2012	• Fix Figure 6.1 - columns & rows were transposed from the real tables.
5.00	December 14, 2012	• 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	• No technical context change: • Update cover with microMIPS logo • Update copyright text. • Update pdf filname.
5.03	August 21, 2012	• 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. .

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.

Revision	Date	Description
5.04	January 15, 2014	LLSC Related Changes • Added ERETNC. New. • Modified SC handling: refined, added, and elaborated cases where SC can fail or was UNPREDICTABLE. XPA Related Changes • Added MTHC0, MFHC0 to access extensions. All new. • Modified MTC0 for MIPS32 to zero out the extended bits which are write-able. This is to support compatibility of XPA hardware with non XPA soft-ware. In pseudo-code, added registers that are impacted. • MTHC0 and MFHC0 - Added RI conditions.
6.0	February 27, 2015	• Release 6 compatible microMIPS. See Section 2.7, "Release 6 of the MIPS Architecture," for instructions that have been added, removed, and recoded.
6.01	June 9, 2015	• Removed the Release 6 NAL instruction; it is not required in microMIPS. • Removed the “Jump and Link Restartability” paragraph from JAL-type instructions; it is not applicable for compact jumps. • Fixed text in jump instructions related to the behavior of ISAMode switch-ing, or lack thereof, in microMIPS. • Removed delay-slot references; all branches/jumps are compact. • Removed references to JALX. • Removed LWXS (bug). • MOVEP: in encoding, changed bit 2 to 1 (bug). • All PC-related instructions: qualify PC with 0x3. Always word aligned. microMIPS only (ADDIUPC, LWPC, AUIPC, ALUIPC). • Release 6 BC: shift-corrected to 1 bit. microMIPS only. • JALRC, JALRC.HB: replace Config1.CA with Config3.ISA (bug). • Added Release 5 TLBINV/TLBINVF (incorrectly excluded from book). • Added Release 6 DVP/EVP instructions. • Added new Release 6 LLX/SCX family instructions. • General opcode map cleanup for consistency with Release 6. Specific opcode map changes: • CACHE, PREF, LL, SC, LLD, SCD, LLX, SCX, LLDX, AND SCDX off-sets changed to 9 bits. • LWC2, SWC2, LDC2, AND SDC2 offsets changed to 11-bits for consis-tency with MIPS Release 6. • Moved BGTZC/BLTZC/BLTC to (5,6) location • Moved BLEZC/BGEZC/BGEC to (5,7) location to free up 16-bit instruction rows for 16-bit instructions (only) • Moved BEQZC/JIC to (0,4) location. • Moved BNEZC/JIALC to (0,5) location bit for differentiating EQ vs NE type made consistent with other branches of this type.

Copyright © 2015 Imagination Technologies LTD. and/or its Affiliated Group Companies. All rights reserved.