MIPS32® Architecture for Programmers: MIPS16e2 Application-Specific Extension Technical Reference Manual

Document Number: MD01172
Revision 01.00
April 26, 2016

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

Chapter 1

Introduction to MIPS16e2

This chapter describes the purpose and key features of the MIPS16e2™ Application-Specific Extension (ASE) to the MIPS32® Architecture. The MIPS16e2 ASE is an enhancement to the previous MIPS16e™ ASE which provides additional instructions to improve the compaction of the code and overall performance.

1.1 Base Architecture Requirements

The MIPS16e2 ASE requires the following base architecture support:

- **The MIPS32 or MIPS64 Architecture**: The MIPS16e2 ASE requires a compliant implementation of the MIPS32 or MIPS64 Architecture.

1.2 Software Detection of the ASE

Software may determine if the MIPS16e2 ASE is implemented by checking the state of the CA2 bit in the *Config5* CP0 register.

1.3 Compliance and Subsetting

There are no instruction subsets of the MIPS16e2 ASE to the MIPS64 Architecture — all MIPS16e2 instructions must be implemented. Specifically, this means that the original MIPS16 ASE is not an allowable subset of the MIPS16e2 ASE. For the MIPS16e2 ASE to the MIPS32 Architecture, the instructions which require a 64-bit processor are not implemented and execution of such an instruction must cause a Reserved Instruction exception.

1.4 MIPS16e2 Overview

The MIPS16e2 ASE allows embedded designs to substantially reduce system cost by reducing overall memory requirements. The MIPS16e2 ASE is compatible with any combination of the MIPS32 or MIPS64 Architectures, and existing MIPS binaries can be run without modification on any embedded processor implementing the MIPS16e2 ASE.

The MIPS16e2 ASE must be implemented as part of a MIPS based host processor that includes an implementation of the MIPS Privileged Resource Architecture, and the other components in a typical MIPS based system.

This volume describes only the new instructions in the MIPS16e2 ASE, and does not include information about any specific hardware implementation such as processor-specific details, because these details may vary with implementation. For this information, please refer to the specific processor's user manual.

1.5 MIPS16e2 ASE Features

The MIPS16e2 ASE includes the following features:

- Includes all MIPS16e instructions — refer to MIPS16e ASE for details
- Allows MIPS16e2 instructions to be intermixed with existing MIPS instruction binaries
- Compatible with the MIPS32 and MIPS64 instruction sets
- Allows switching between MIPS16e2 and 32-bit MIPS Mode
- Supports 8, 16, 32, and 64-bit data types (64-bit only in conjunction with MIPS64)
- Defines eight general-purpose registers, as well as a number of special-purpose registers
- Defines special instructions to increase code density (Extend, PC-relative instructions)

The MIPS16e2 ASE contains some instructions that are available on MIPS64 host processors only. These instructions must cause a Reserved Instruction exception on 32-bit processors, or on 64-bit processors on which 64-bit operations have not been enabled.

1.6 MIPS16e2 Register Set

The MIPS16e2 register set is listed in Table 1.1 and Table 1.2. This register set is a true subset of the register set available in 32-bit mode; the MIPS16e2 ASE can directly access 8 of the 32 registers available in 32-bit mode.

Table 1.1 lists the general purpose registers, and Table 1.2 lists the MIPS16e2 special-purpose registers, including *PC*.

The MIPS16e2 ASE also contains two move instructions that provide access to all 32 general-purpose registers.

Table 1.1 MIPS16e2 General-Purpose Registers

MIPS16e Register Encoding[1]	32-Bit MIPS Register Encoding[2]	Symbolic Name (From *ArchDefs.h*)[3]	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
7	7	a3	General-purpose register

Table 1.1 MIPS16e2 General-Purpose Registers

MIPS16e Register Encoding[1]	32-Bit MIPS Register Encoding[2]	Symbolic Name (From *ArchDefs.h*)[3]	Description
N/A	24	t8	MIPS16e2 *Condition Code* register; implicitly referenced by the BTEQ, BTNE, CMP, CMPI, SLT, SLTU, SLTI, and SLTIU instructions
N/A	28	gp	Global pointer register
N/A	29	sp	Stack pointer register
N/A	31	ra	Return address register

1. "0-7" correspond to the register's MIPS16e2 binary encoding and show how that encoding relates to the MIPS registers. "0-7" never refer to the registers, except within the binary MIPS16e2 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 MIPS16e2 binary encoding for this register is 001.
2. General registers not shown in the above table are not accessible through the MIPS16e2 instruction set, except by using the Move instructions. The MIPS16e2 Move instructions can access all 32 general-purpose registers.
3. The MIPS16e2 condition code register is referred to as T, t8, or $24 throughout this document, depending on the context. All three names refer to the same physical register.

Table 1.2 MIPS16e2 Special-Purpose Registers

Symbolic Name	Purpose
PC	Program counter. The PC-relative Add and Load instructions 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.

1.7 MIPS16e2 ISA Modes

This section describes the following:

- the ISA modes available in the architecture, page 7
- the purpose of the *ISA Mode* field, page 8

1.7.1 Modes Available in the MIPS16e2 Architecture

There are two ISA modes defined in the MIPS16e2 Architecture, as follows:

- MIPS 32-bit mode (32-bit instructions)

- MIPS16e mode (16-bit instructions)

1.7.2 Defining the ISA Mode Field

The *ISA Mode* bit controls the type of code that is executed, as follows:

Table 1.3 ISA Mode Bit Encodings

Encoding	Mode
0b0	MIPS 32-bit mode. In this mode, the processor executes 32-bit MIPS instructions.
0b1	MIPS16e mode. In this mode, the processor executes MIPS16e instructions.

1.7.3 Switching Between Modes When an Exception Occurs

When an exception occurs (including a Reset exception), the *ISA Mode* bit is cleared so that exceptions are handled by 32-bit code.

The ISA Mode in which the processor was running at the time that the exception occurred is visible to software as bit 0 of the Coprocessor 0 register in which the restart address is stored (*EPC*, *ErrorEPC*, or *DEPC*). See the description of these instructions in Volume III for a complete description of this process.

After the processor switches to 32-bit mode following a Reset exception, the processor starts execution at the 32-bit mode Reset exception vector.

1.8 MIPS16e2 Instruction Summaries

This section describes the various instruction categories and then summarizes the MIPS16e2 instructions included in each category. Extensible instructions are also identified.

There are six instruction categories:

- **Loads and Stores** — These instructions move data between memory and the GPRs.
- **Immediate** —These instructions perform arithmetic, logical, and shift operations on immediate values.
- **Move** — These instructions perform move operations.
- **Special** — This category includes the Break and Extend instructions. Break transfers control to an exception handler, and Extend enlarges the *immediate* field of the next instruction.
- **Enable and Disable** — This category enables and disables hardware such as interrupt generation, multi-threading, and virtual processors.

Tables 1.4 through 1.8 list the MIPS16e2 instruction set.

Table 1.4 MIPS16e2 Load and Store Instructions

Mnemonic	Instruction	Instruction Always Extendible?	Implemented Only on MIPS64 Processors?
LB	Load Byte (global pointer)	Yes	No
LBU	Load Byte Unsigned (global pointer)	Yes	No
LH	Load Halfword (global pointer)	Yes	No
LHU	Load Halfword Unsigned (global pointer)	Yes	No
LW	Load Word (global pointer)	Yes	No
LWL	Load Word Left	Yes	No
LWR	Load Word Right	Yes	No
SB	Store Byte (global pointer)	Yes	No
SH	Store Halfword (global pointer)	Yes	No
SW	Store Word (global pointer)	Yes	No
SWL	Store Word Left	Yes	No
SWR	Store Word Right	Yes	No
SC	Store Conditional	Yes	No

Table 1.5 MIPS16e2 Immediate Instructions

Mnemonic	Instruction	Instruction Always Extendible?	Implemented Only on MIPS64 Processors?
ADDIU	Add Immediate Unsigned (global pointer)	Yes	No
ANDI	Logical AND immediate	Yes	No
ORI	Logical OR immediate	Yes	No
XORI	Logical Exclusive OR immediate	Yes	No
LL	Load Linked Word Immediate	Yes	No
LUI	Load Upper Immediate Extended	Yes	No

Table 1.6 MIPS16e2 Move Instructions

Mnemonic	Instruction	Instruction Always Extendible?	Implemented Only on MIPS64 Processors?
MFC0	Move from Coprocessor 0	Yes	No
MTC0	Move to Coprocessor 0	Yes	No
MOVN	Move Conditional on Not Equal to Zero Extended	Yes	No
MOVN	Move Zero Conditional on Not Equal to Zero Extended	Yes	No
MOVZ	Move Conditional on Equal to Zero Extended	Yes	No
MOVZ	Move Zero Conditional on Not Equal to Zero Extended	Yes	No
MOVTN	Move Conditional on T Not Equal to Zero Extended	Yes	No
MOVTN	Move Zero Conditional on T Not Equal to Zero Extended	Yes	No
MOVTZ	Move Conditional on T Equal to Zero Extended	Yes	No
MOVTZ	Move Zero Conditional on T Equal to Zero Extended	Yes	No

Table 1.7 MIPS16e2 Special Instructions

Mnemonic	Instruction	Instruction Always Extendible?	Implemented Only on MIPS64 Processors?
CACHE	Perform Cache Operation Extended	Yes	No
EHB	Execution Hazard Barrier Extended	Yes	No
EXT	Extract Bit Field Extended	Yes	No
INS	Insert Bit Field Extended	Yes	No
INS0	Insert Bit Field 0 Extended	Yes	No
PAUSE	Wait for the LLBit to Clear Extended	Yes	No
PREF	Prefetch Extended	Yes	No
RDHWR	Read Hardware Register Extended	Yes	No
SYNC	Synchronize Shared Memory Extended	Yes	No

Table 1.8 MIPS16e2 Enable and Disable Instructions

Mnemonic	Instruction	Instruction Always Extendible?	Implemented Only on MIPS64 Processors?
DI	Disable Interrupts	Yes	No
DMT	Disable Multi-threading	Yes	No
DVPE	Disable Virtual Processor Execution	Yes	No
EI	Enable Interrupts	Yes	No
EMT	Enable Multi-threading	Yes	No
EVPE	Enable Virtual Processor Execution	Yes	No

1.9 MIPS16e2 Instruction Formats

This section defines the format[1] for each MIPS16e2 instruction type and includes formats for both normal and extended instructions.

Every MIPS16e2 instruction consists of 16 bits aligned on a halfword boundary. All variable subfields in an instruction format (such as `rx`, `ry`, `rz`, and `immediate`) are shown in lowercase letters.

The two instruction subfields `op` and `funct` have constant values for specific instructions. These values are given in their uppercase mnemonic names. For example, `op` is LB in the Load Byte instruction; `op` is RRR and `function` is ADDU in the Add Unsigned instruction.

Definitions for the fields that appear in the instruction formats are summarized in Table 1.9.

Table 1.9 MIPS16e2 Instruction Fields

Field	Definition
funct or f	Function field

1. As used here, the term *format* means the layout of the MIPS16e2 instruction word.

immediate or imm	9- or 16-bit immediate field
rx	3-bit source or destination register specifier
ry	3-bit source or destination register specifier
rb	3-bit source or destination register specifier
sel	3-bit select field
op	5-bit opcode field

1.10 MIPS16e2 Instruction Stream Organization and Endianness

The instruction halfword is 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
- 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 all extended instructions, since they consist of two 16-bit halfwords. For a 16-bit-instruction sequence, instructions are placed in memory so that an LH instruction with the PC as an argument fetches the instruction independent of system endianness.

1.11 MIPS16e2 Instruction Fetch Restrictions

When the processor is running in MIPS16e2 mode and fetch address is in uncacheable memory, certain restrictions apply to the width of each instruction fetch. Under these circumstances, the processor never fetches more than an aligned word during each instruction fetch. It is UNPREDICTABLE whether the processor fetches a single aligned word, or two aligned halfwords during each instruction fetch.

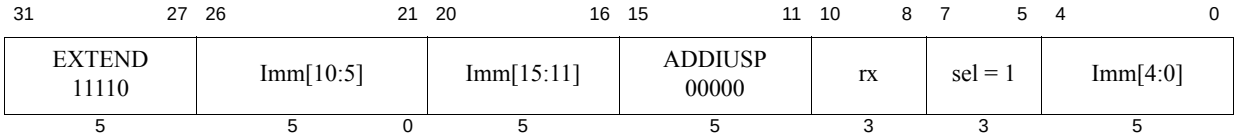
1.12 Xlat Usage

Many of the instructions in this document use the Xlat function when defining the destination register. The Xlat function translates the MIPS16e register field index to the correct 32-bit MIPS physical register index. It is used to assure that a value of 0b000 in a MIPS16e register field maps to GPR 16, and a value of 0b001 maps to GPR 17. All other values (0b010 through 0b111) map directly.

Chapter 2

MIPS16e2 Instruction Descriptions

This chapter describes the MIPS16e2 instructions in alphabetical order.



Format: `ADDIU rx, gp, immediate` **MIPS16e2**

Purpose: Add Immediate Unsigned Word (3-Operand, GP-Relative, Extended)

To add a constant to the global pointer.

Description: `GPR[rx] ← GPR[gp] + immediate`

The 16-bit *immediate* is sign-extended and then added to the contents of GPR 28 to form a 32-bit result. The result is placed in GPR rx.

No integer overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

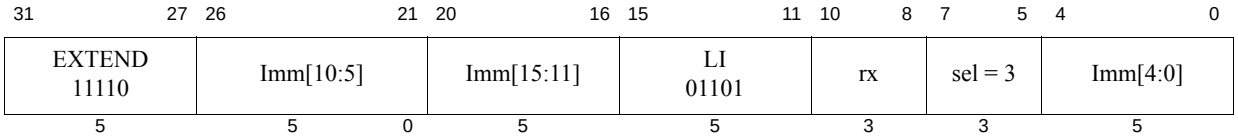
```
temp ← GPR[28] + sign_extend(immediate)
GPR[XLat[rx]] ← temp
```

Exceptions:

None

Programming Notes:

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



Format: `ANDI rx, immediate` **MIPS16e2**

Purpose: AND Immediate Extended

To do a bitwise logical AND with a constant.

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

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

Restrictions:

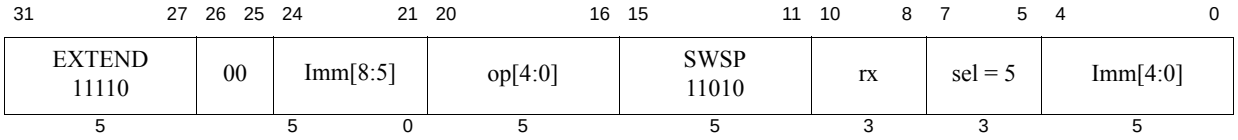
Unpredictable prior to MIPS16e2.

Operation:

```
GPR[XLat[rx]] ← GPR[XLat[rx]] and zero_extend(immediate)
```

Exceptions:

None



Format: `CACHE op, immediate(rx)` **MIPS16e2**

Purpose: Perform Cache Operation Extended

To perform the cache operation specified by the *op* field.

Description:

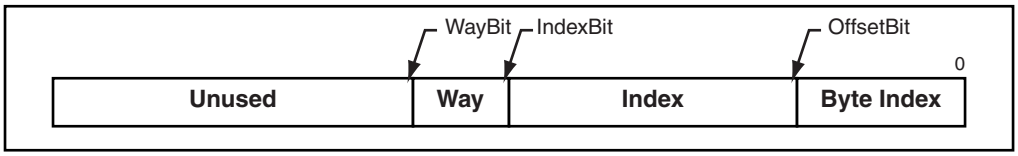
The 9-bit *immediate* value is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

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.

Usage of Address Fields to Select Index and Way



A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur on any operation. For index

operations (where the address is used to index the cache but need not match the cache tag), software 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 [17:16] of the instruction specify the cache on which to perform the operation, as follows:

Encoding of Bits[17:16] of CACHE Instruction

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

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

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 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.

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
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

Encoding of Bits [20:18] of the CACHE Instruction *(continued)*

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
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
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.

Encoding of Bits [20:18] of the CACHE Instruction *(continued)*

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance Implemented
0b111	I, D	Fetch and Lock	Address	If the cache does not contain the specified address, fill it from memory, performing a write-back if required. 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

Restrictions:

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

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

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

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

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

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

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

Operation:

```
vAddr ← GPR[XLat[rx]] + sign_extend(immediate)
(pAddr, uncached) ← AddressTranslation(vAddr, DataReadReference)
CacheOp(op, vAddr, pAddr)
```

Exceptions:

TLB Refill Exception.

TLB Invalid Exception

Coprocessor Unusable Exception

Address Error Exception

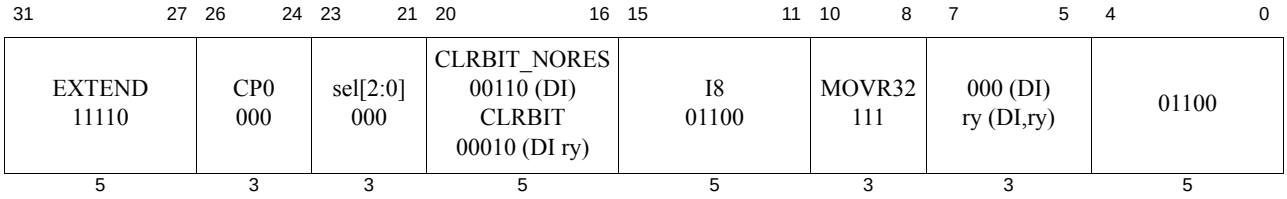
Cache Error Exception

Bus Error Exception

Programming Notes:

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

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



Format: `DI` **MIPS16e2**

`DI ry` **MIPS16e2**

Purpose: Disable Interrupts Extended

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[ry] ← Status; Status`$_{IE}$ `← 0`

The current value of the *Status* register is loaded into general register *ry*. The Interrupt Enable (IE) bit in the *Status* register is then cleared.

Restrictions:

Unpredictable prior to MIPS16e2. If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation — DI:

The following operation pertains to the *DI* instruction.

```
Status_IE ← 0
```

Operation — DI ry:

The following operation pertains to the *DI ry* instruction.

```
data ← Status
GPR[XLat[ry]] ← data
Status_IE ← 0
```

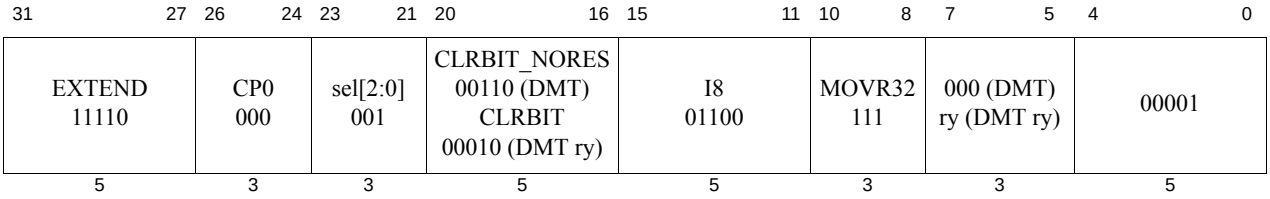
Exceptions:

Coprocessor Unusable

Programming Notes:

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

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



Format: `DMT` **MIPS16e2**

`DMT ry` **MIPS16e2**

Purpose: Disable Multi-Threaded Execution Extended

To return the previous value of the *VPEControl* register and disable multi-threaded execution. If DMT is specified without an argument, GPR *r0* is implied, which discards the previous value of the *VPEControl* register.

Description: `GPR[ry] ← VPEControl; VPEControl`$_{TE}$ `← 0`

The current value of the *VPEControl* register isloaded into general register *ry*. The Threads Enable (*TE*) bit in the *VPEControl* register is then cleared, suspending concurrent execution of instruction streams other than that which issues the DMT. This is independent of any per-TC halted state.

Restrictions:

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

In implementations that do not implement the MT Module, this instruction results in a Reserved Instruction Exception. Unpredictable prior to MIPS16e2.

Operation — DMT:

The following operation pertains to the *DMT* instruction.

```
VPEControl_TE ← 0
```

Operation — DMT ry:

The following operation pertains to the *DMT ry* instruction.

```
data ← VPEControl
GPR[XLat[ry]] ← sign_extend(data)
VPEControl_TE ← 0
```

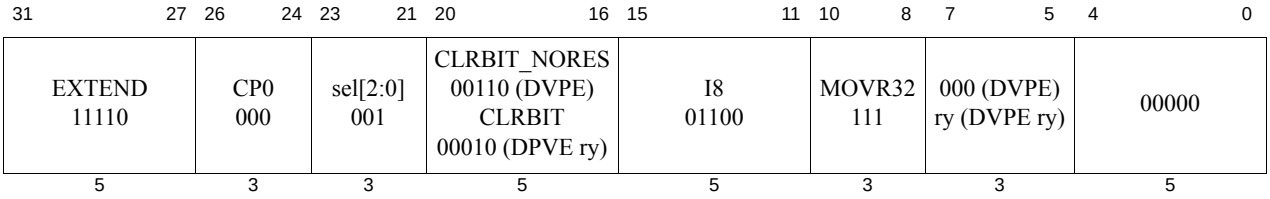
Exceptions:

Coprocessor Unusable
Reserved Instruction (Implementations that do not include the MT Module)

Programming Notes:

The effects of this instruction are identical to those accomplished by the sequence of reading *VPEControl* into a GPR, clearing the *TE* bit to create a temporary value in a second GPR, and writing that value back to *VPEControl*. Unlike the multiple instruction sequence, however, the DMT instruction does not consume a temporary register, and cannot be aborted by an interrupt or exception.

The effect of a DMT instruction may not be instantaneous. An instruction hazard barrier, e.g., JR.HB, is required to guarantee that all other threads have been suspended. If a DMT instruction is followed in the same instruction stream by an MFC0 or MFTR from the *VPEControl* register, a JALR.HB, JR.HB, EHB, or ERET instruction must be issued between the DMT and the read of *VPEControl* to guarantee that the new state of *TE* will be accessed by the read.



Format: `DVPE` **MIPS16e2**
`DVPE ry`

Purpose: Disable Virtual Processor Execution Extended

To return the previous value of the *MVPControl* register and disable multi-VPE execution. If DVPE is specified without an argument, GPR r0 is implied, which discards the previous value of the *MVPControl* register.

Description: `GPR[ry] ← MVPControl; MVPControl`$_{EVP}$ `← 0`

The current value of the *MVPControl* register isloaded into general register *ry*. The Enable Virtual Processors (*EVP*) bit in the *MVPControl* register is then cleared, suspending concurrent execution of instruction streams other than the instruction stream that issues the DVPE.

Restrictions:

Unpredictable prior to MIPS16e2. If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled. If the VPE executing the instruction is not a Master VPE, with the *MVP* bit of the *VPEConf0* register set, the *EVP* bit is unchanged by the instruction.

In implementations that do not implement the MT Module, this instruction results in a Reserved Instruction Exception.

Operation — DVPE:

The following operation pertains to the *DVPE* instruction.

```
if(VPEConf0MVP = 0) then
   MVPControlEVP ← 0
endif
```

Operation — DVPE ry:

The following operation pertains to the *DVPE ry* instruction.

```
data ← MVPControl
GPR[XLat[ry]] ← data
   if(VPEConf0MVP = 0) then
      MVPControlEVP ← 0
   endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction (Implementations that do not include the MT Module)

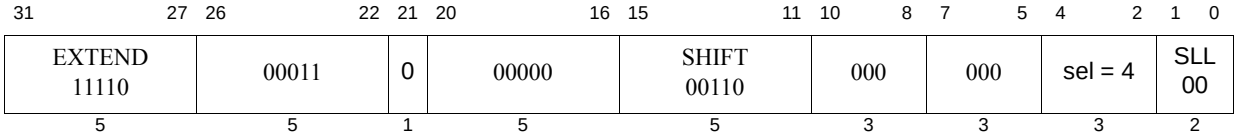
Programming Notes:

The effects of this instruction are identical to those accomplished by the sequence of reading *MVPControl* into a GPR, clearing the *EVP* bit to create a temporary value in a second GPR, and writing that value back to *MVPControl*. Unlike the multiple instruction sequence, however, the DVPE instruction does not consume a temporary register, and cannot

be aborted by an interrupt or exception, nor by the scheduling of a different instruction stream.

The effect of a DVPE instruction may not be instantaneous. An instruction hazard barrier, e.g., JR.HB, is required to guarantee that all other TCs have been suspended.

If a DVPE instruction is followed in the same instruction stream by an MFC0 or MFTR from the *MVPControl* register, a JALR.HB, JR.HB, EHB, or ERET instruction must be issued between the DVPE and the read of *MVPControl* to guarantee that the new state of *EVP* will be accessed by the read.



Format: `EHB` **MIPS16e2**

Purpose: Execution Hazard Barrier Extended

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, even if the EHB is executed in the delay slot of a branch or jump. The EHB instruction does not clear instruction hazards—such hazards are cleared by the JALR.HB, JR.HB, and ERET instructions.

Restrictions:

Unpredictable prior to MIPS16e2.

Operation:

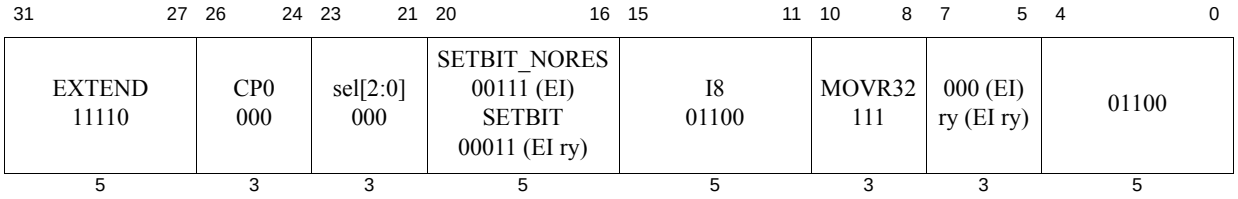
```
ClearExecutionHazards()
```

Exceptions:

None

Programming Notes:

This instruction resolves all execution hazards.



Format: `EI` **MIPS16e2**

`EI ry` **MIPS16e2**

Purpose: Enable Interrupts Extended

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[ry] ← Status; Status`$_{IE}$ `← 1`

The current value of the *Status* register is loaded into general register *ry*. The Interrupt Enable (*IE*) bit in the *Status* register is then set.

Restrictions:

Unpredictable prior to MIPS16e2. If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation — EI:

The following operation pertains to the *EI* instruction.

```
Status_IE ← 1
```

Operation — EI ry:

The following operation pertains to the *EI ry* instruction.

```
data ← Status
GPR[XLat[ry]] ← data
Status_IE ← 1
```

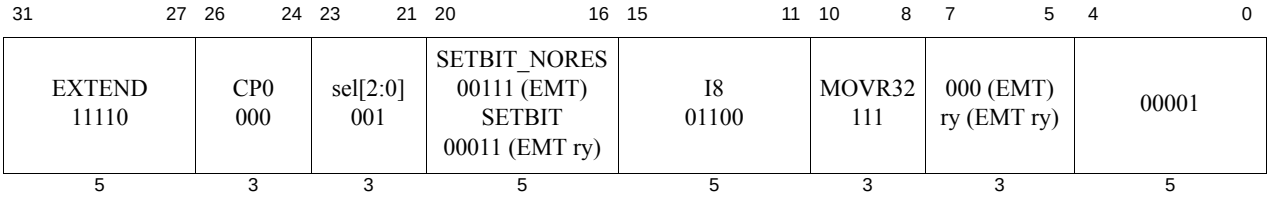
Exceptions:

Coprocessor Unusable
Reserved Instruction

Programming Notes:

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

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



Format: `EMT` **MIPS16e2**

`EMT ry` **MIPS16e2**

Purpose: Enable Multi-Threaded Execution Extended

To return the previous value of the *VPEControl* register and to enable multi-threaded execution. If EMT is specified without an argument, GPR *r0* is implied, which discards the previous value of the *VPEControl* register.

Description: `GPR[ry] ← VPEControl; VPEControl`$_{TE}$ `← 1`

The current value of the *VPEControl* register isloaded into general register *ry*. The Threads Enable (*TE*) bit in the *VPEControl* register is then set, allowing multiple instruction streams to execute concurrently.

Restrictions:

Unpredictable prior to MIPS16e2. If access to Coprocessor 0 is not enabled, a **Coprocessor Unusable Exception** is signaled.

In implementations that do not implement the MT Module, this instruction results in a **Reserved Instruction Exception**.

Operation — EMT:

The following operation pertains to the *EMT* instruction.

```
VPEControl_TE ← 1
```

Operation — EMT ry:

The following operation pertains to the *EMT ry* instruction.

```
data ← VPEControl
GPR[XLat[ry]] ← sign_extend(data)
VPEControl_TE ← 1
```

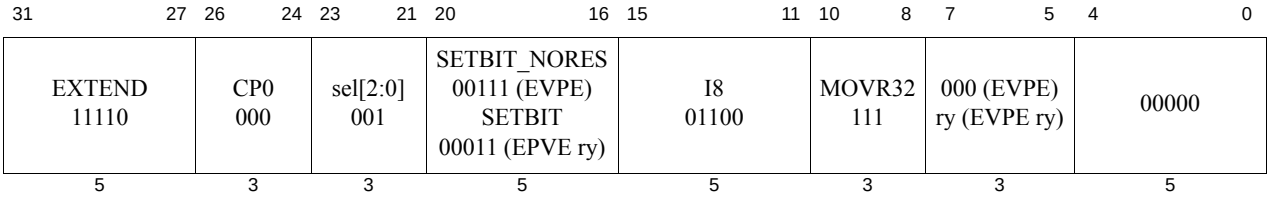
Exceptions:

Coprocessor Unusable
Reserved Instruction (Implementations that do not include the MT Module)

Programming Notes:

The effects of this instruction are identical to those accomplished by the sequence of reading *VPEControl* into a GPR, setting the *TE* bit to create a temporary value in a second GPR, and writing that value back to *VPEControl*. Unlike the multiple instruction sequence, however, the EMT instruction does not consume a temporary register, and cannot be aborted by an interrupt or exception.

If an EMT instruction is followed in the same instruction stream by an MFC0 or MFTR from the *VPEControl* register, a JALR.HB, JR.HB, EHB, or ERET instruction must be issued between the EMT and the read of *VPEControl* to guarantee that the new state of *TE* will be accessed by the read.



Format: `EVPE` **MIPS16e2**

`EVPE ry`

Purpose: Enable Virtual Processor Execution Extended

To return the previous value of the *MVPControl* register and enable multi-VPE execution. If EVPE is specified without an argument, GPR r0 is implied, which discards the previous value of the *MVPControl* register.

Description: `GPR[ry] ← MVPControl; MVPControl`$_{EVP}$ `← 1`

The current value of the *MVPControl* register isloaded into general register *ry*. The Enable Virtual Processors (*EVP*) bit in the *MVPControl* register is then set, enabling concurrent execution of instruction streams on all non-inhibited Virtual Processing Elements (VPEs) on a processor.

Restrictions:

Unpredictable prior to MIPS16e2. If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled. If the VPE executing the instruction is not a Master VPE, with the *MVP* bit of the *VPEConf0* register set, the *EVP* bit is unchanged by the instruction.

In implementations that do not implement the MT Module, this instruction results in a Reserved Instruction Exception.

Operation — EVPE:

The following operation pertains to the *EVPE* instruction.

```
if(VPEConf0MVP = 1) then
   MVPControlEVP ← 1
endif
```

Operation — EVPE ry:

The following operation pertains to the *EVPE ry* instruction.

```
data ← MVPControl
GPR[XLat[ry]] ← data
   if(VPEConf0MVP = 1) then
      MVPControlEVP ← 1
   endif
```

Exceptions:

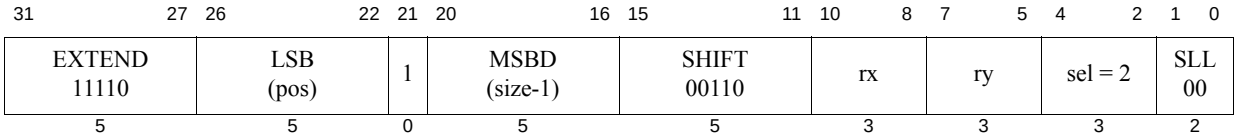
Coprocessor Unusable
Reserved Instruction (Implementations that do not include the MT Module)

Programming Notes:

The effects of this instruction are identical to those accomplished by the sequence of reading *MVPControl* into a GPR, setting the *EVP* bit to create a temporary value in a second GPR, and writing that value back to *MVPControl*. Unlike the multiple instruction sequence, however, the EVPE instruction does not consume a temporary register, and cannot

be aborted by an interrupt or exception, nor by the scheduling of a different instruction stream.

If an EVPE instruction is followed in the same instruction stream by an MFC0 or MFTR from the *MVPControl* register, a JALR.HB, JR.HB, EHB, or ERET instruction must be issued between the EVPE and the read of *MVPControl* to guarantee that the new state of *EVP* will be accessed by the read.



Format: `EXT ry, rx, pos, size` **MIPS16e2**

Purpose: Extract Bit Field Extended

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

Description: `GPR[ry] ← ExtractField(GPR[rx], msbd, lsb)`

The bit field starting at bit *pos* and extending for *size* bits is extracted from GPR *rx* and stored zero-extended and right-justified in GPR *ry*. 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 *ry*), in instruction bits 20..16, and *lsb* (least significant bit of the source field in GPR *rx*), in instruction bits 26..22, as follows:

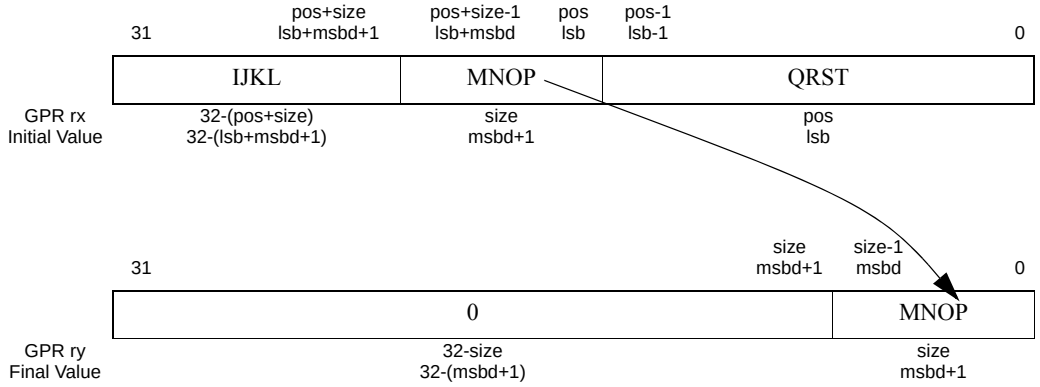
```
msbd ← size-1
lsb ← pos
```

The values of *pos* and *size* must satisfy all of the following relations:

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

The following figure shows the symbolic operation of the instruction.

Operation of the EXT Instruction



Restrictions:

In implementations prior to MIPS16e2, this instruction yields unpredictable results. It would typically be executed as an SLL instruction. The operation is **UNPREDICTABLE** if *lsb*+*msbd* > 31.

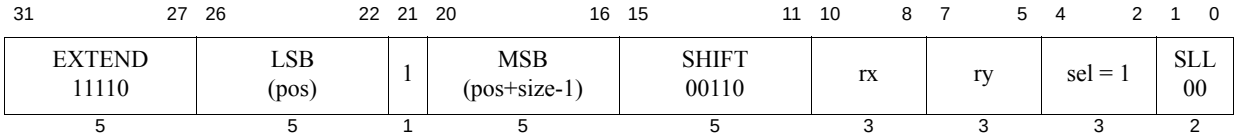
Operation:

```
if (lsb + msbd) > 31)then
   UNPREDICTABLE
endif
temp ← 0^(32-(msbd+1)) || GPR[XLat[rx]]msbd+lsb..lsb
```

```
GPR[XLat[ry]] ← temp
```

Exceptions:

None



Format: `INS ry, rx, pos, size` **MIPS16e2**

Purpose: Insert Bit Field Extended

To merge a right-justified bit field from GPR *rx* into a specified field in GPR *ry*.

Description: `GPR[ry] ← InsertField(GPR[ry], GPR[rx], msb, lsb)`

The right-most *size* bits from GPR *rx* are merged into the value from GPR *ry* starting at bit position *pos*. The result is placed back in GPR *ry*. 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 20..16, and *lsb* (least significant bit of the field), in instruction bits 26..22, as follows:

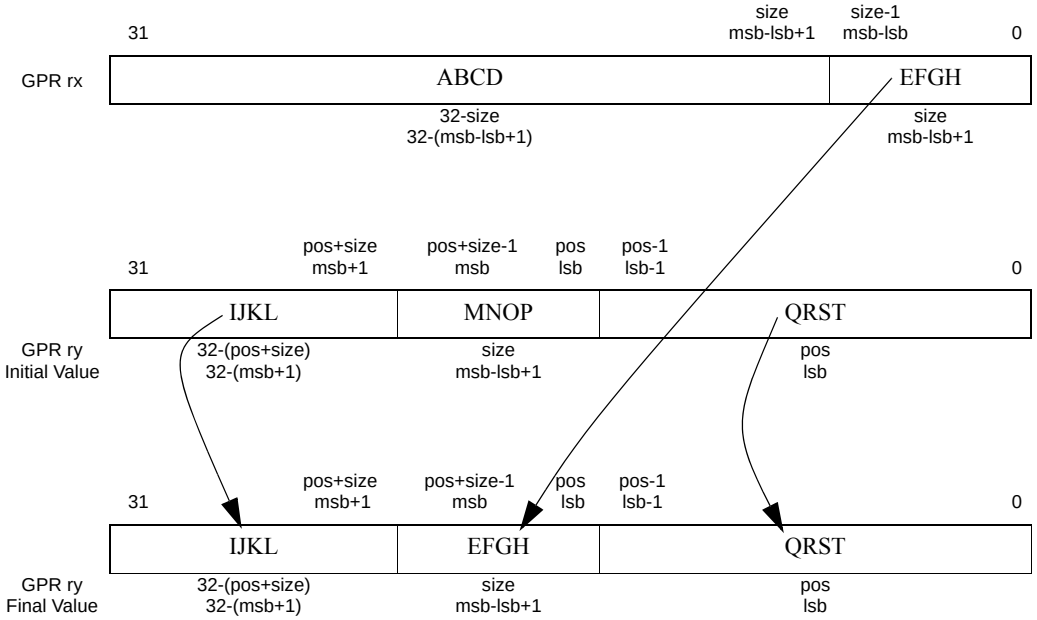
```
msb ← pos+size-1
lsb ← pos
```

The values of *pos* and *size* must satisfy all of the following relations:

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

The following figure shows the symbolic operation of the instruction.

Operation of the INS Instruction



Restrictions:

In implementations prior to MIPS16e2, this instruction yields unpredictable results. Typically, it would be silently

executed as an SLL instruction. The operation is **UNPREDICTABLE** if *lsb* > *msb*.

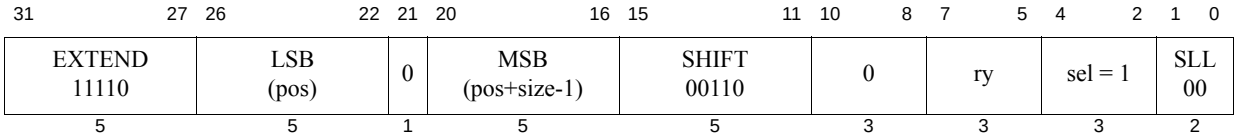
Operation:

```
if lsb > msb)then
   UNPREDICTABLE
endif
```

$GPR[XLat[ry]] \leftarrow GPR[XLat[ry]]_{31..msb+1} \,||\, GPR[XLat[rx]]_{msb-lsb..0} \,||\, GPR[XLat[ry]]_{lsb-1..0}$

Exceptions:

None



Format: `INS ry, $0, pos, size` **MIPS16e2**

Purpose: Insert Bit Field 0 Extended

To merge bits with a value of zero into a specified field in GPR *ry*.

Description: `GPR[ry] ← InsertField(GPR[ry], msb, lsb)`

Size bits with a value of zero are merged into the value from GPR *ry* starting at bit position *pos*. The result is placed back in GPR *ry*. 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 20..16, and *lsb* (least significant bit of the field), in instruction bits 26..22, as follows:

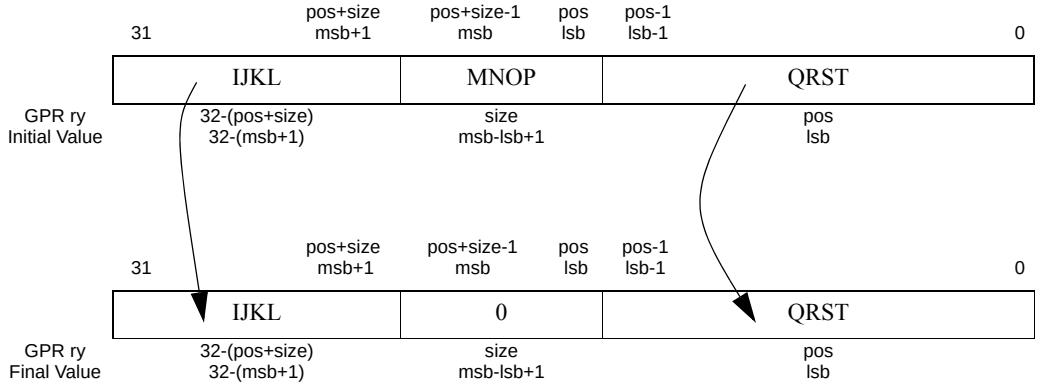
```
msb ← pos+size-1
lsb ← pos
```

The values of *pos* and *size* must satisfy all of the following relations:

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

The following figure shows the symbolic operation of the instruction.

Operation of the INS Instruction



Restrictions:

In implementations prior to MIPS16e2, this instruction yield unpredictable results. It would typically be performed as an SLL instruction.

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

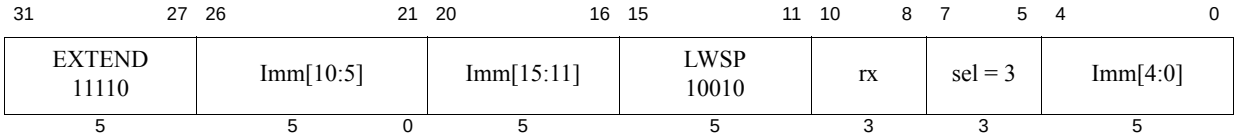
Operation:

```
if lsb > msb)then
   UNPREDICTABLE
endif
```

```
GPR[XLat[ry]] ← GPR[XLat[ry]]31..msb+1 || GPR[XLat[ry]]msb-lsb..0 || GPR[XLat[ry]]lsb-1..0
```

Exceptions:

None



Format: `LB rx, immediate(gp)` **MIPS16e2**

Purpose: Load Byte (GP-relative) Extended

To load a byte from memory as a signed value.

Description: `GPR[rx] ← memory[GPR[gp] + immediate]`

The 16-bit *immediate* value is sign-extended, then added to the contents of GPR *28* to form the effective address. The contents of the byte at the memory location specified by the effective address are sign-extended and loaded into GPR *rx*.

Restrictions:

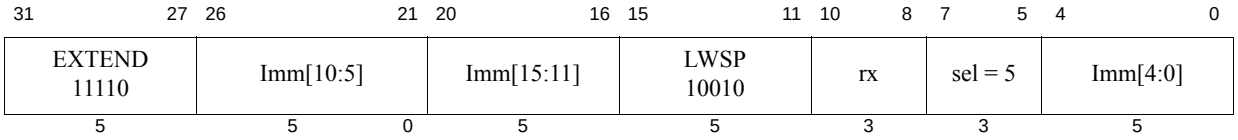
Unpredictable prior to MIPS16e2.

Operation:

```
vAddr ← sign_extend(immediate) + GPR[28]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_1..0 xor BigEndianCPU^2
GPR[Xlat(rx)] ← sign_extend(memword_7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error



Format: `LBU rx, immediate(gp)` **MIPS16e2**

Purpose: Load Byte Unsigned (GP-relative) Extended

To load a byte from memory as an unsigned value

Description: `GPR[rx] ← memory[GPR[gp] + immediate]`

The 16-bit *immediate* value is sign-extended, then added to the contents of GPR *28* to form the effective address. The contents of the byte at the memory location specified by the effective address are zero-extended and loaded into GPR *rx*.

Restrictions:

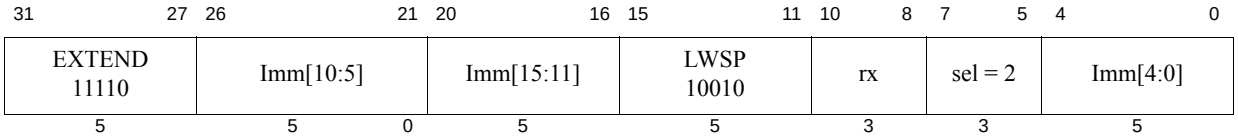
Unpredictable prior to MIPS16e2.

Operation:

```
vAddr ← sign_extend(immediate) + GPR[28]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
memword ← LoadMemory (CCA, BTE, pAddr, vAddr, DATA)
byte ← vAddr_1..0 xor BigEndianCPU^2
GPR[Xlat(rx)] ← zero_extend(memword_7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error



Format: `LH rx, immediate(gp)` **MIPS16e2**

Purpose: Load Halfword (GP-relative) Extended

To load a halfword from memory as a signed value.

Description: `GPR[rx] ← memory[GPR[gp] + immediate]`

The 16-bit *immediate* value is sign-extended, then added to the contents of GPR *28* to form the effective address. The contents of the halfword at the memory location specified by the effective address are sign-extended and loaded into GPR *rx*.

Restrictions:

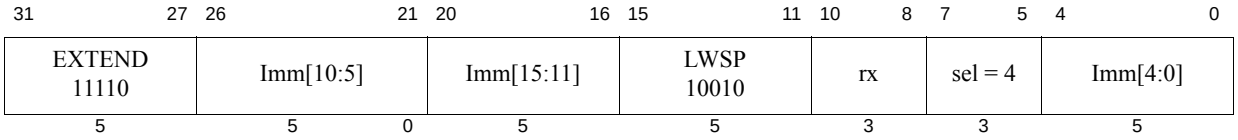
Unpredictable prior to MIPS16e2. The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(immediate) + GPR[28]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr1..0 xor (BigEndianCPU || 0)
GPR[Xlat(rx)] ← sign_extend(memword15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error



Format: `LHU rx, immediate(gp)` **MIPS16e2**

Purpose: Load Halfword Unsigned Extended

To load a halfword from memory as an unsigned value.

Description: `GPR[rx] ← memory[GPR[gp] + immediate]`

The 16-bit *immediate* value is sign-extended, then added to the contents of GPR *28* to form the effective address. The contents of the halfword at the memory location specified by the effective address are zero-extended and loaded into GPR *rx*.

Restrictions:

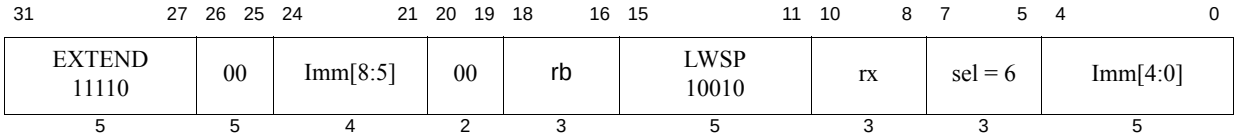
Unpredictable prior to MIPS16e2. The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(immediate) + GPR[28]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr1..0 xor (BigEndianCPU || 0)
GPR[Xlat(rx)] ← zero_extend(memword15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error



Format: `LL rx, immediate(rb)` **MIPS16e2**

Purpose: Load Linked Word Immediate

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

Description: `GPR[rx] ← memory[GPR[rb] + immediate]`

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 *rx*. The 9-bit signed *immediate* value is added to the contents of GPR *rb* 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:

Unpredictable prior to MIPS16e2. The addressed location must be synchronizable by all processors and I/O devices sharing the location; if it is not, the result is **UNPREDICTABLE**. Which storage is synchronizable is a function of both CPU and system implementations. See the documentation of the SC instruction for the formal definition.

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

Operation:

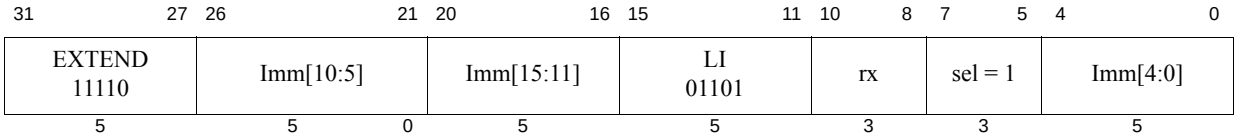
```
vAddr ← sign_extend(immediate) + GPR[XLat[rb]]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[XLat[rx]]← memword
LLbit ← 1
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch

Programming Notes:

MIPS16e2 implements a 9-bit immediate value as the offset.



Format: `LUI rx, immediate` **MIPS16e2**

Purpose: Load Upper Immediate Extended

To load a constant into the upper half of a word.

Description: `GPR[rx] ← 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 *rx*.

Restrictions:

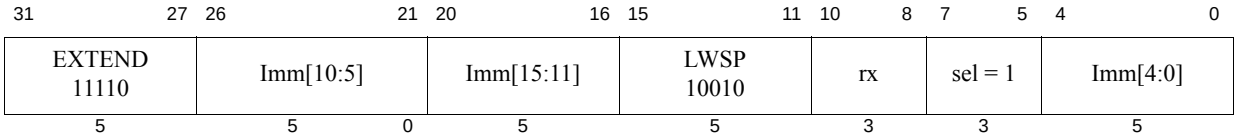
Unpredictable prior to MIPS16e2.

Operation:

GPR[XLat[rx]] ← `immediate || 0`^{16}

Exceptions:

None



Format: `LW rx, immediate(gp)` **MIPS16e2**

Purpose: Load Word (GP-Relative, Extended)

To load a GP-relative word from memory as a signed value.

Description: `GPR[rx] ← memory[GPR[gp] + immediate]`

The 16-bit *immediate* value is sign-extended, then added to the contents of GPR *28* to form the effective address. The contents of the word at the memory location specified by the effective address are loaded into GPR *rx*.

Restrictions:

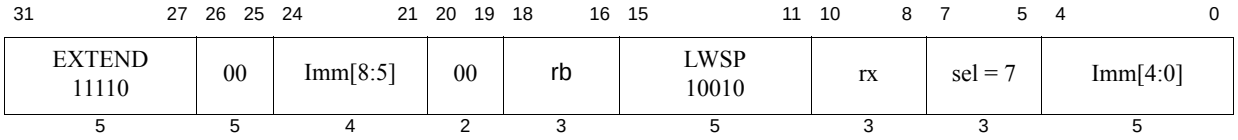
Unpredictable prior to MIPS16e2. The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(immediate) + GPR[28]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[Xlat(rx)] ← memword
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error



Format: `LWL rx, immediate(rb)` **MIPS16e2**

Purpose: Load Word Left Extended

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

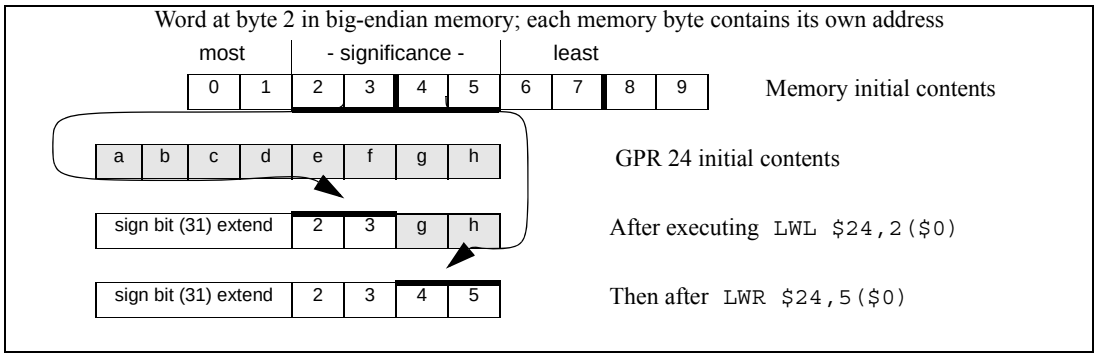
Description: `GPR[rx] ← GPR[rx] MERGE memory[GPR[rb] + immediate]`

The 9-bit signed *immediate* value is added to the contents of GPR *rb* to form an effective address *(EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

The most-significant 1 to 4 bytes of *W* is in the aligned word containing the *EffAddr*. This part of *W* is loaded into the most-significant (left) part of the word in GPR *rx*. The remaining least-significant part of the word in GPR *rx* is unchanged.

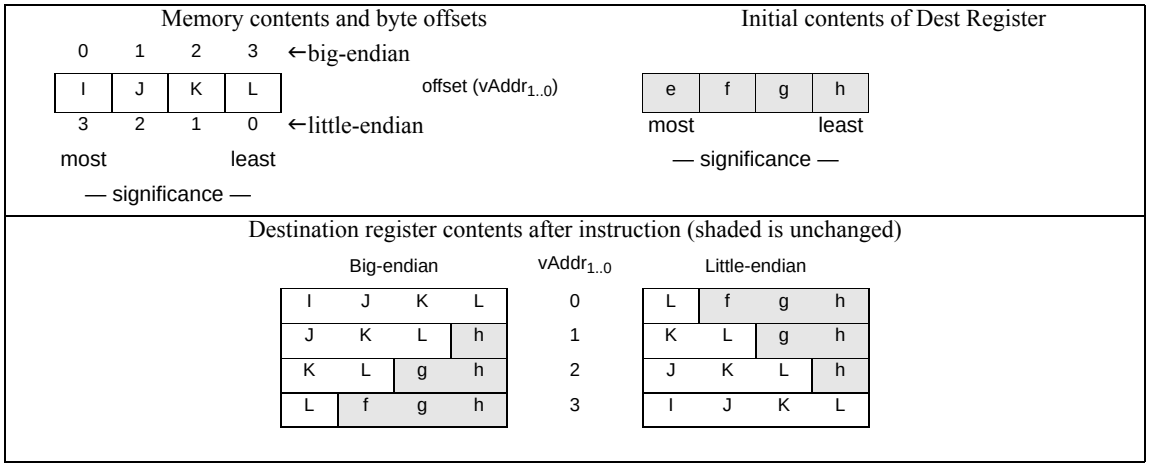
The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2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

Unaligned Word Load Using LWL and LWR



The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address ($vAddr_{1..0}$), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.

Bytes Loaded by LWL Instruction



Restrictions:

Unpredictable prior to MIPS16e2.

Operation:

```
vAddr ← sign_extend(immediate) + GPR[XLat[rb]]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
if BigEndianMem = 0 then
   pAddr ← pAddr_PSIZE-1..2 || 0^2
endif
byte ← vAddr_1..0 xor BigEndianCPU^2
memword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp ← memword_7+8*byte..0 || GPR[XLat[rx]]_23-8*byte..0
GPR[XLat[rx]] ← temp
```

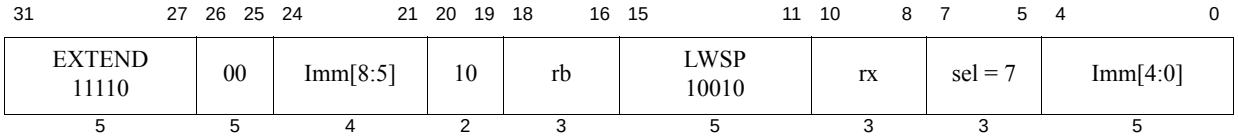
Exceptions:

None

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

Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.



Format: `LWR rx, immediate(rb)` **MIPS16e2**

Purpose: Load Word Right Extended

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

Description: `GPR[rx] ← GPR[rx] MERGE memory[GPR[rb] + immediate]`

The 9-bit signed *immediate* value is added to the contents of GPR *rb* to form an effective address *(EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

A part of *W* (the least-significant 1 to 4 bytes) is in the aligned word containing *EffAddr*. This part of *W* is loaded into the least-significant (right) part of the word in GPR *rx*. The remaining most-significant part of the word in GPR *rx* is unchanged.

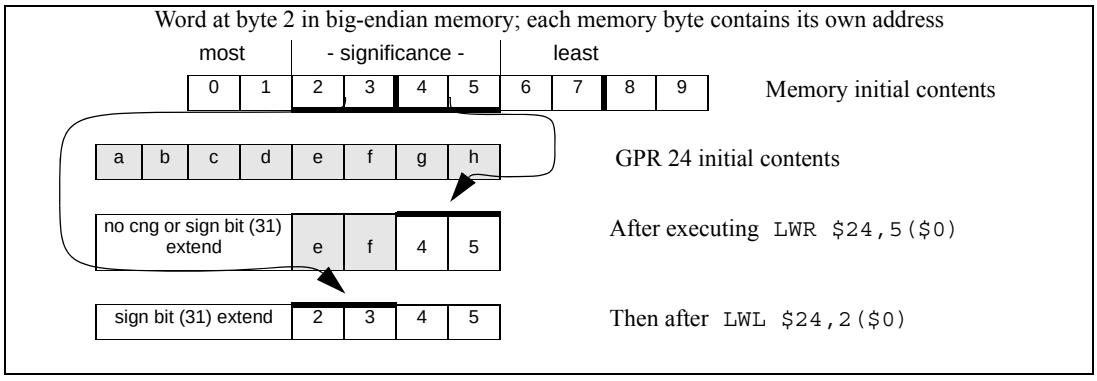
Executing both LWR and LWL, in either order, delivers a sign-extended word value in the destination register.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the least-significant byte at 5.

1. LWR loads these 2 bytes into the right part of the destination register.

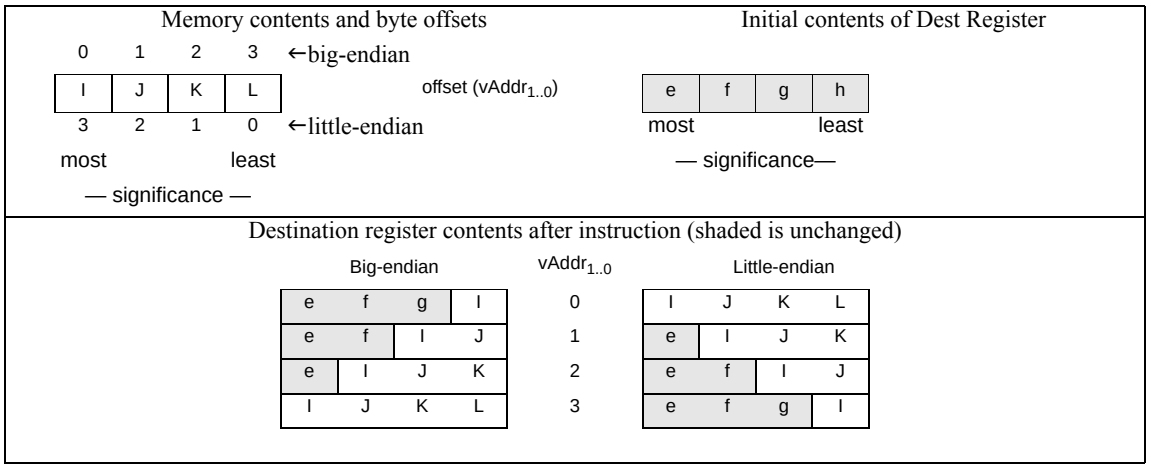
2. The complementary LWL loads the remainder of the unaligned word.

Unaligned Word Load Using LWL and LWR



The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address ($vAddr_{1..0}$), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.

Bytes Loaded by LWR Instruction



Restrictions:

Unpredictable prior to MIPS16e2.

Operation:

```
vAddr ← sign_extend(immediate) + GPR[XLat[rb]]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
if BigEndianMem = 0 then
   pAddr ← pAddr_PSIZE-1..2 || 0^2
endif
byte ← vAddr_1..0 xor BigEndianCPU^2
memword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp ← memword_31..32-8*byte || GPR[XLat[rx]]_31-8*byte..0
GPR[XLat[rx]] ← temp
```

Exceptions:

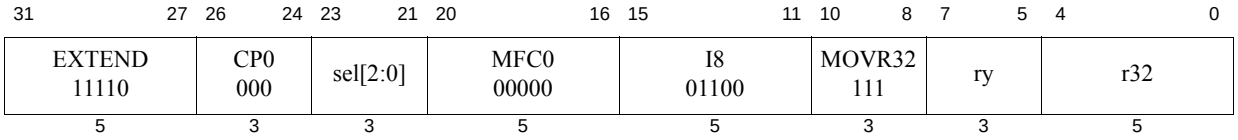
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

Historical Information:

In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.



Format: `MFC0 ry, r32, sel` **MIPS16e2**

Purpose: Move from Coprocessor 0 Extended

To move the contents of a coprocessor 0 register to a general register.

Description: `GPR[ry] ← CPR[0,r32,sel]`

The contents of the coprocessor 0 register specified by the combination of *r32* and *sel* are loaded into general register *ry*. Not all coprocessor 0 registers support the *sel* field. In those instances, the *sel* field must be zero.

Restrictions:

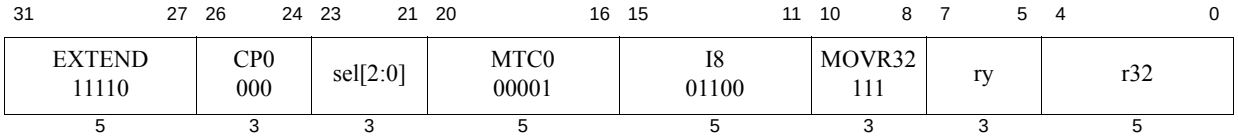
The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *r32* and *sel*.

Operation:

```
reg = r32
if IsCoprocessorRegisterImplemented(0, reg, sel) then
   data ← CPR[0, reg, sel]
   GPR[XLat[ry]] ← data
else
   if ArchitectureRevision() ≥ 6 then
      GPR[XLat[ry]] ← 0
   else
      UNDEFINED
   endif
endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction



Format: `MTC0 ry, r32, sel` **MIPS16e2**

Purpose: Move to Coprocessor 0 Extended

To move the contents of a general register to a coprocessor 0 register.

Description: `CPR[0, r32, sel] ← GPR[ry]`

The contents of general register ry are loaded into the coprocessor 0 register specified by the combination of *r32* and *sel*. Not all coprocessor 0 registers support the *sel* field. In those instances, the *sel* field must be set to zero.

Restrictions:

Unpredictable prior to MIPS16e2. The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *r32* and *sel*.

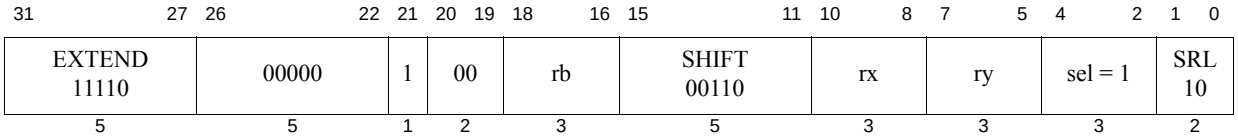
Operation:

```
data ← GPR[XLat[ry]]
reg ← r32
if IsCoprocessorRegisterImplemented (0, reg, sel) then
   CPR[0,reg,sel] ← data
          if (Config5MVH = 1) then
              // The most-significant bit may vary by register. Only supported
              // bits should be written 0. Extended LLAddr is not written with 0s,
              // as it is a read-only register. BadVAddr is not written with 0s, as
              // it is read-only
              if (Config3LPA = 1) then
                  if (reg,sel = EntryLo0 or EntryLo1) then CPR[0,reg,sel]63:32 = 032
                  if (reg,sel = MAAR) then CPR[0,reg,sel]63:32 = 032 endif
                      // TagLo is zeroed only if the implementation-dependent bits
                      // are writeable
                  if (reg,sel = TagLo) then CPR[0,reg,sel]63:32 = 032 endif
                  if (Config3VZ = 1) then
                      if (reg,sel = EntryHi) then CPR[0,reg,sel]63:32 = 032 endif
                  endif
          endif
   endif
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction



Format: `MOVZ rx, rb, ry` **MIPS16e2**

Purpose: Move Conditional on Equal to Zero Extended

To conditionally move a GPR after testing a GPR value.

Description: `if GPR[ry] = 0 then GPR[rx] ← GPR[rb]`

If the value in GPR *ry* is equal to zero, then the contents of GPR *rb* are placed into GPR *rx*.

Restrictions:

In implementations prior to MIPS16e2, this instruction yielded unpredictable results. It would typically be executed as an SRL instruction.

Operation:

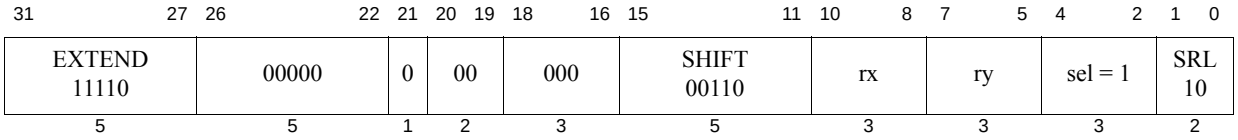
```
if GPR[XLat[ry]] = 0 then
   GPR[XLat[rx]] ← GPR[XLat[rb]]
endif
```

Exceptions:

None

Programming Notes:

The zero value tested might be the *condition false* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions or a boolean value read from memory.



Format: `MOVZ rx, $0, ry` **MIPS16e2**

Purpose: Move Zero Conditional on Equal to Zero Extended

To conditionally zero a GPR after testing a GPR value.

Description: `if GPR[ry] = 0 then GPR[rx] ← 0`

If the value in GPR *ry* is equal to zero, then GPR *rx* is written with the value of 0.

Restrictions:

In implementations prior to MIPS16e2, this instruction yielded unpredictable results. It would typically be exexcuted as an SRL instruction.

Operation:

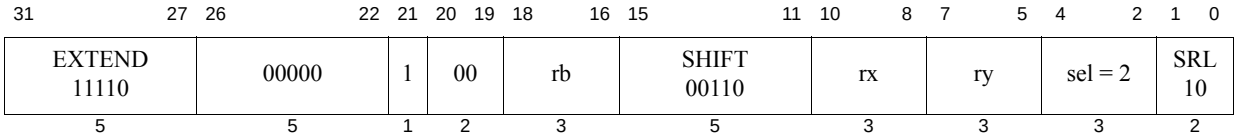
```
if GPR[XLat[ry]] = 0 then
   GPR[XLat[rx]] ← 0
endif
```

Exceptions:

None

Programming Notes:

The zero value tested might be the *condition false* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions or a boolean value read from memory.



Format: `MOVN rx, rb, ry` **MIPS16e2**

Purpose: Move Conditional on Not Equal to Zero Extended

To conditionally move a GPR after testing a GPR value.

Description: `if GPR[ry] ≠ 0 then GPR[rx] ← GPR[rb]`

If the value in GPR *ry* is not equal to zero, then the contents of GPR *rb* are placed into GPR *rx*.

Restrictions:

In implementations prior to MIPS16e2, this instruction yielded unpredictable results. It would typically be executed as an SRL instruction.

Operation:

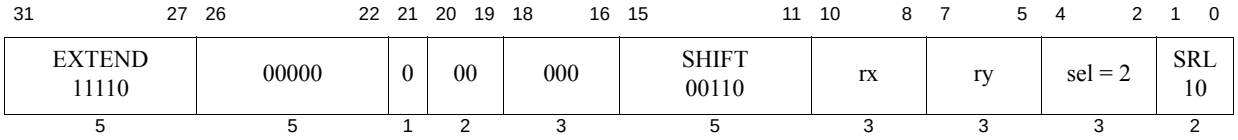
```
if GPR[XLat[ry]] ≠ 0 then
   GPR[XLat[rx]] ← GPR[XLat[rb]]
endif
```

Exceptions:

None

Programming Notes:

The non-zero value tested might be the *condition true* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions or a boolean value read from memory.



Format: `MOVN rx, $0, ry` **MIPS16e2**

Purpose: Move Zero Conditional on Not Equal to Zero Extended

To conditionally zero a GPR after testing a GPR value.

Description: `if GPR[ry] ≠ 0 then GPR[rx] ← 0`

If the value in GPR *ry* is not equal to zero, GPR rx is written with the value of 0.

Restrictions:

In implementations prior to MIPS16e2, this instruction yielded unpredictable results. It would typically be exexcuted as an SRL instruction.

Operation:

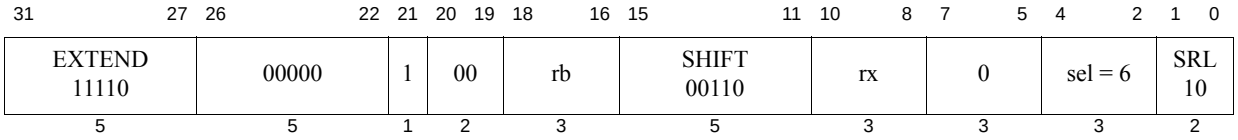
```
if GPR[XLat[ry]] ≠ 0 then
   GPR[XLat[rx]] ← 0
endif
```

Exceptions:

None

Programming Notes:

The non-zero value tested might be the *condition true* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions or a boolean value read from memory.



Format: `MOVTN rx, rb` **MIPS16e2**

Purpose: Move Conditional on T Not Equal to Zero Extended

Test special register T and then conditionally move a GPR.

Description: `If T != 0, then GPR[rx] ← GPR[rb]`

If the value in GPR[24] is not equal to 0, the contents of GPR *rb* are placed into GPR *rx*.

Restrictions:

In implementations prior to MIPS16e2, this instruction yielded unpredictable results. It would typically be executed as an SRL instruction.

Operation:

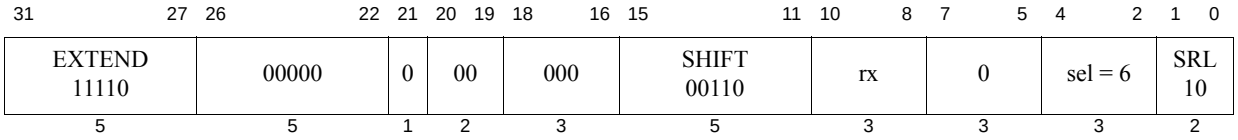
```
if GPR[24] != 0, then
   GPR[XLat[rx]] ← GPR[XLat[rb]]
endif
```

Exceptions:

None

Programming Notes:

The non-zero value tested might be the *condition true* result from the CMP or CMPI comparison instructions or a boolean value read from memory.



Format: `MOVTN rx, $0` **MIPS16e2**

Purpose: Move Zero Conditional on T Not Equal to Zero Extended

Test special register T and then conditionally move a GPR after testing a GPR value.

Description: `If T != 0, then GPR[rx] ← 0`

If the value in GPR[24] is not equal to 0, GPR *rx* is written with the value 0.

Restrictions:

In implementations prior to MIPS16e2, this instruction yielded unpredictable results. It would typically be executed as an SRL instruction.

Operation:

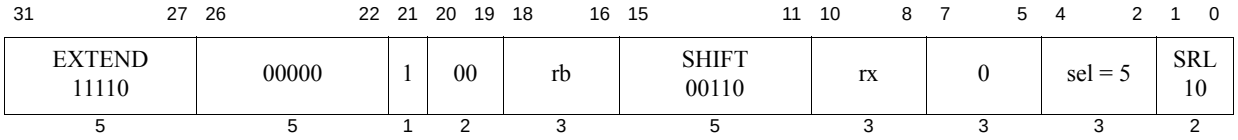
```
if GPR[24] != 0, then
   GPR[XLat[rx]] ← 0
endif
```

Exceptions:

None

Programming Notes:

The non-zero value tested might be the *condition true* result from the CMP or CMPI comparison instructions or a boolean value read from memory.



Format: `MOVTZ rx, rb` **MIPS16e2**

Purpose: Move Conditional on T Equal to Zero Extended

To test special register T and then conditionally move a GPR.

Description: `If T = 0, then GPR[rx] ← GPR[rb]`

If the value in GPR[24] is equal to 0, the contents of GPR *rb* are placed into GPR *rx*.

Restrictions:

In implementations prior to MIPS16e2, this instruction yielded unpredictable results. It would typically be exexcuted as an SRL instruction.

Operation:

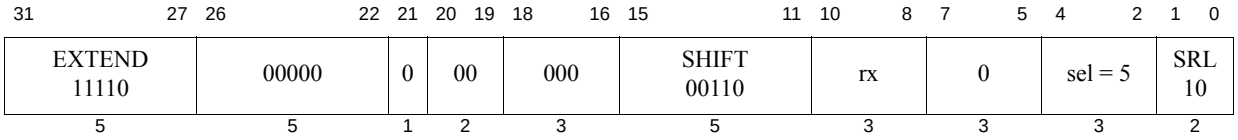
```
if GPR[24] = 0 then
   GPR[XLat[rx]] ← GPR[XLat[rb]]
endif
```

Exceptions:

None

Programming Notes:

The zero value tested might be the *condition false* result from the CMP or CMPI comparison instructions or a boolean value read from memory.



Format: `MOVTZ rx, $0` **MIPS16e2**

Purpose: Move Zero Conditional on T Equal to Zero Extended

To test special register T and then conditionally move a GPR after testing a GPR value.

Description: `If T = 0, then GPR[rx] ← 0`

If the value in GPR[24] is equal to 0, GPR *rx* is written with the value 0.

Restrictions:

Unpredictable prior to MIPS16e2.

Operation:

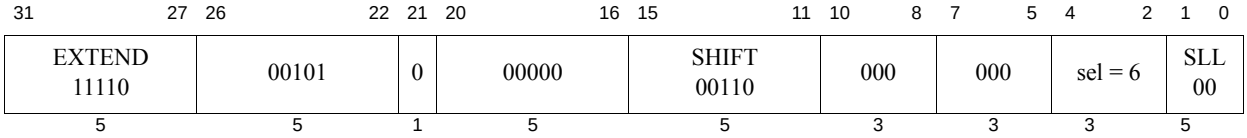
```
if GPR[24] = 0 then
   GPR[XLat[rx]] ← 0
endif
```

Exceptions:

In implementations prior to MIPS16e2, this instruction yielded unpredictable results. It would typically be executed as an SRL instruction.

Programming Notes:

The zero value tested might be the *condition false* result from the CMP or CMPI comparison instructions or a boolean value read from memory.



Format: `PAUSE` **MIPS16e2**

Purpose: Wait for the LLBit to Clear Extended

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.

Restrictions:

Unpredictable prior to MIPS16e2. 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.

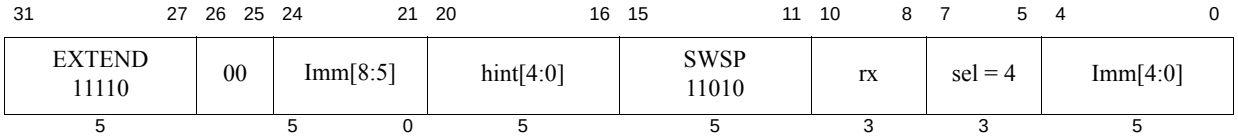
An example use of the PAUSE instruction is included in the following example:

```
acquire_lock:
   ll     v0, 0(a0)              /* Read software lock, set hardware lock */
   bnez   v0, acquire_lock_retry: /* Branch if software lock is taken */
   addiu  v0, v0, 1              /* Set the software lock */
   sc     v0, 0(a0)              /* Try to store the software lock */
```

```
    bnez v0, 10f              /* Branch if lock acquired successfully */
    sync
acquire_lock_retry:
    pause                        /* Wait for LLBIT to clear before retry */
    b      acquire_lock          /* and retry the operation */
10:

    Critical region code

release_lock:
    sync
    li     t1, 0                 /* Release software lock, clearing LLBIT */
    sw     t1, 0(a0)             /* for any PAUSEd waiters */
```



Format: `PREF hint,immediate(rx)` **MIPS16e2**

Purpose: Prefetch Extended

To move data between memory and cache.

Description: `prefetch_memory(GPR[rx] + immediate)`

PREF adds the signed *immediate* to the contents of GPR *rx* 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 write-back 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.

Values of *hint* Field for PREF Instruction

Value	Name	Data Use and Desired Prefetch Action
0	load	Use: Prefetched data is expected to be read (not modified). Action: Fetch data as if for a load.
1	store	Use: Prefetched data is expected to be stored or modified. Action: Fetch data as if for a store.

Values of *hint* Field for PREF Instruction *(continued)*

Value	Name	Data Use and Desired Prefetch Action
2	L1 LRU hint	Pre-Release 6: Reserved for Architecture. Release 6: Implementation-dependent. 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	L2 operation	Pre-Release 6: Reserved for Architecture. 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	L3 operation	Pre-Release 6: Reserved for Architecture. 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. Release 6: 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”)	Pre-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. Release 6: 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. Release 6: These hints are not implemented in the Release 6 architecture and generate a Reserved Instruction exception (RI).

Values of *hint* Field for PREF Instruction *(continued)*

Value	Name	Data Use and Desired Prefetch Action
30	PrepareForStore	Pre-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. Release 6: 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. Release 6: This hint is not implemented in the Release 6 architecture and generates a Reserved Instruction exception (RI).

Restrictions:

Unpredictable prior to MIPS16e2.

Operation:

```
vAddr ← GPR[Xlat[rx]] + sign_extend(immediate)
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Bus Error, Cache Error

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:

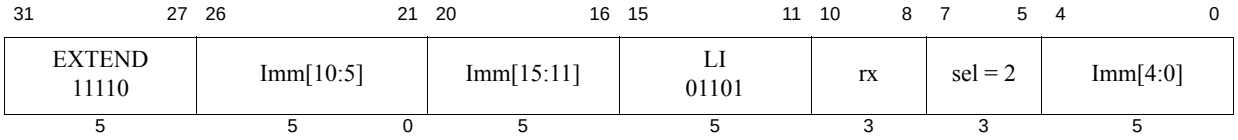
Prefetch cannot move data to or from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. A prefetch may be used using an address pointer before the validity of the pointer is determined without worrying about an addressing exception.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREF instruction. Typically, this only occurs in systems which have high-reliability requirements.

Prefetch operations have no effect on cache lines that were previously locked with the CACHE instruction.

Hint field encodings whose function is described as “streamed” or “retained” convey usage intent from software to hardware. Software should not assume that hardware will always prefetch data in an optimal way. If data is to be truly retained, software should use the Cache instruction to lock data into the cache.



Format: `ORI rx, immediate` **MIPS16e2**

Purpose: Or Immediate Extended

To do a bitwise logical OR with a constant.

Description: `GPR[rx] ← GPR[rx] OR immediate`

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

Restrictions:

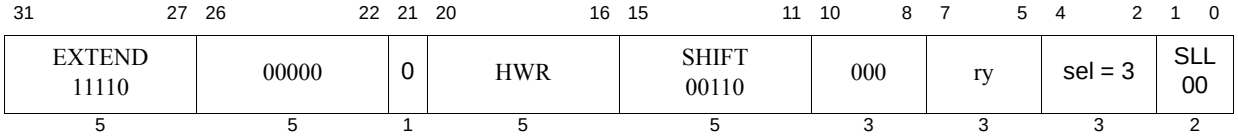
Unpredictable prior to MIPS16e2.

Operations:

```
GPR[XLat[rx]] ← GPR[Xlat[rx]] or zero_extend(immediate)
```

Exceptions:

None



Format: `RDHWR ry,HWR` **MIPS16e2**

Purpose: Read Hardware Register Extended

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[ry] ← HWR[HWR]`

If access is allowed to the specified hardware register, the contents of the register specified by *SHIFT* is loaded into general register *ry* Access control for each register is selected by the bits in the coprocessor 0 *HWREna* register.

The available hardware registers, and the encoding of the *rd* field for each, are shown below.

RDHWR Register Numbers

Register Number (*HWR* 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	Rsv	Reserved.
5	XNP	Indicates support for Release 6 Double-Width LLX/SCX family of instructions. If set to 1, then LLX/SCX family of instructions is not present, otherwise present in the implementation. In absence of hardware support for double-width or extended atomics, user software may emulate the instruction's behavior through other means. See $Config5_{XNP}$.
6-28		These registers numbers are reserved for future architecture use. Access results in a Reserved Instruction Exception.

RDHWR Register Numbers

Register Number (*HWR* Value)	Mnemonic	Description
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.

Restrictions:

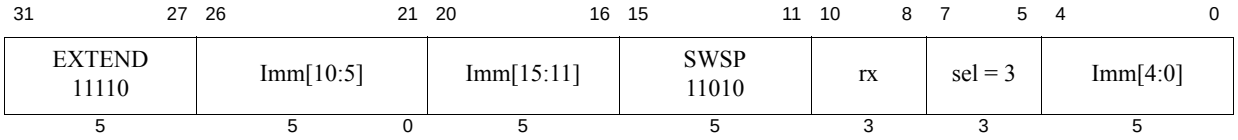
Unpredictable prior to MIPS16e2. Access to the specified hardware register is enabled if Coprocessor 0 is enabled, or if the corresponding bit is set in the *HWREna* register. If access is not allowed or the register is not implemented, a Reserved Instruction Exception is signaled.

Operation:

```
case HWR
       0: temp ← EBase_CPUNum
   1: temp ← SYNCI_StepSize()
   2: temp ← Count
   3: temp ← CountResolution()
   5: temp ← XNP
   29: temp ← UserLocal
   30: temp ← Implementation-Dependent-Value
   31: temp ← Implementation-Dependent-Value
   otherwise: SignalException(ReservedInstruction)
endcase
   else
   endif
GPR[Xlat[ry]] ← temp
```

Exceptions:

Reserved Instruction



Format: `SB rx, immediate(gp)` **MIPS16e2**

Purpose: Store Byte (GP-relative) Extended

To store a byte to memory.

Description: `memory[GPR[gp] + immediate] ← GPR[rx]`

The 16-bit *immediate* value is sign-extended, then added to the contents of GPR *28* to form the effective address. The least-significant byte of GPR *rx* is stored at the effective address.

Restrictions:

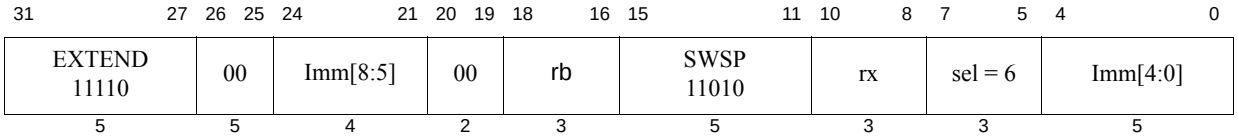
Unpredictable prior to MIPS16e2.

Operation:

```
vAddr ← sign_extend(immediate) + GPR[28]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
bytesel ← vAddr_1..0 xor BigEndianCPU^2
dataword ← GPR[Xlat[rx]]_31-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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



Format: `SC rx, immediate(rb)` **MIPS16e2**

Purpose: Store Conditional Word Extended

To store a word to memory to complete an atomic read-modify-write.

Description: `if atomic_update then memory[GPR[rb] + immediate] ← GPR[rx], GPR[rx] ← 1 else GPR[rx] ← 0`

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations on synchronizable memory locations.

The 32-bit word in GPR *rx* is conditionally stored in memory at the location specified by the aligned effective address. The signed *immediate* value is added to the contents of GPR *rb* 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 *rx* is stored to memory at the location specified by the aligned effective address.
- A one, indicating success, is written into GPR *rx*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *r*x

If the following event occurs between the execution of LL and SC, the SC fails:

- 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.

Furthermore, an SC must always compare its address against that of the LL. An SC will fail if the aligned address of the SC does not match that of the preceeding LL.

A load that executes on the processor executing the LL/SC sequence to the block of synchronizable physical memory containing the word, will not cause the SC to fail (if $Config5_{LLB}$=1; else such a load may cause the SC to fail).

If any of the events listed below occurs between the execution of LL and SC, the SC may fail where it could have succeeded, i.e., success is not predictable. Portable programs should not cause any of these events.

- A load or store executed on the processor executing the LL and SC that is not to the block of synchronizable physical memory containing the word. (The load or store may cause a cache eviction between the LL and SC that results in SC failure. The load or store does not necessarily have to occur between the LL and SC.)
- Any prefetch that is executed on the processor executing the LL and SC sequence (due to a cache eviction between the LL and SC).
- A non-coherent store executed between an LL and SC sequence to the block of synchronizable physical memory containing the word.
- The instructions executed starting with the LL and ending with the SC do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

CACHE operations that are local to the processor executing the LL/SC sequence will result in unpredictable behaviour of the SC if executed between the LL and SC, that is, they may cause the SC to fail where it could have succeeded. Non-local CACHE operations (address-type with coherent CCA) may cause an SC to fail on either the local processor or on the remote processor in multiprocessor or multi-threaded systems. This definition of the effects of CACHE operations is mandated if $Config5_{LLB}$=1. If $Config5_{LLB}$=0, then CACHE effects are implementation-dependent.

The following conditions must be true or the result of the SC is not predictable—the SC may fail or succeed (if $Config5_{LLB}$=1, then either success or failure is mandated, else the result is **UNPREDICTABLE**):

- Execution of SC must have been preceded by execution of an LL instruction.
- An RMW sequence executed without intervening events that would cause the SC to fail must use the same address in the LL and SC. The address is the *same* if the virtual address, physical address, and cacheability & coherency attribute are identical.

Restrictions:

Unpredictable prior to MIPS16e2. 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(immediate) + GPR[Xlat[rb]]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword← GPR[Xlat[rx]]
if LLbit then
   StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
endif
GPR[Xlat[rx]] ← 0^31 || LLbit
LLbit ← 0 // if Config5_LLB=1, SC always clears LLbit regardless of address match.
```

Exceptions:

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

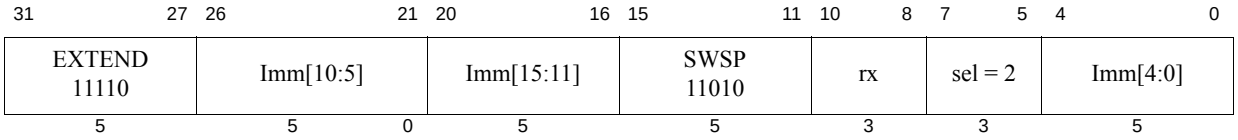
Programming Notes:

LL and SC are used to atomically update memory locations, as shown below.

```
L1:
   LL     a1, (a0)  # load counter
   ADDIU  v0, a1, 1 # increment
   SC     v0, (a0)  # try to store, checking for atomicity
   BEQ    v0, 0, L1 # if not atomic (0), try again
   NOP              # branch-delay slot
```

Exceptions between the LL and SC cause SC to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LL and SC function on a single processor for *cached noncoherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.



Format: `SH rx, immediate(gp)` **MIPS16e2**

Purpose: Store Halfword (GP-relative)

To store a halfword to memory.

Description: `memory[GPR[gp] + immediate] ← GPR[rx]`

The 16-bit *immediate* value is sign-extended, and then added to the contents of GPR *28* to form the effective address. The least-significant halfword of GPR *rx* is stored at the effective address.

Restrictions:

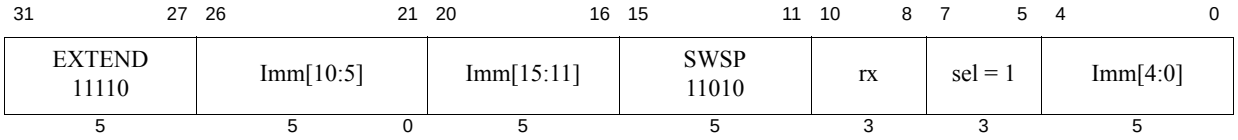
Unpredictable prior to MIPS16e2. The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(immediate) + GPR[28]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1..2 || (pAddr1_1..0 xor (ReverseEndian || 0))
bytesel ← vAddr1_1..0 xor (BigEndianCPU || 0)
dataword ← GPR[Xlat[rx]]_31-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, HALFWORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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



Format: `SW rx, immediate(gp)` **MIPS16e2**

Purpose: Store Word (GP-relative) Extended

To store a word to memory.

Description: `memory[GPR[gp] + immediate] ← GPR[rx]`

The 16-bit *immediate* value is sign-extended, then added to the contents of GPR *28* to form the effective address. The contents of GPR *rx* are stored at the effective address.

Restrictions:

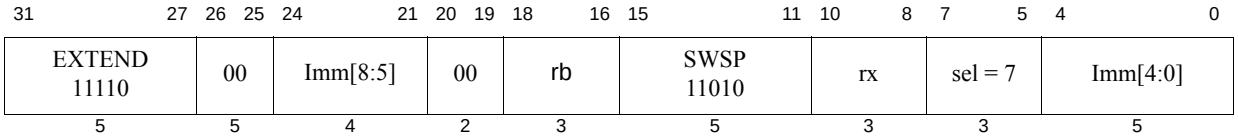
Unpredictable prior to MIPS16e2. The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(immediate) + GPR[28]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[Xlat(rx)]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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



Format: `SWL rx, immediate(rb)` **MIPS16e2**

Purpose: Store Word Left Extended

To store the most-significant part of a word to an unaligned memory address.

Description: `memory[GPR[rb] + immediate] ← GPR[rx]`

The 9-bit signed *immediate* value is added to the contents of GPR *rb* to form an effective address *(EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

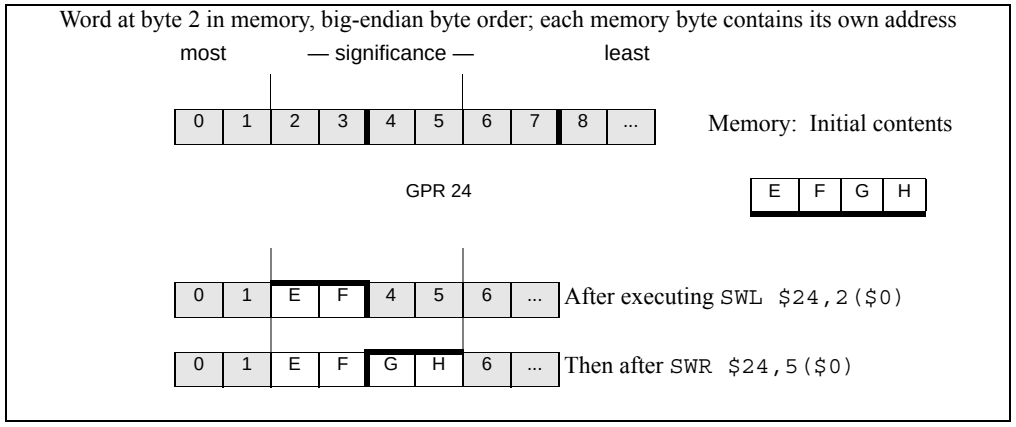
A part of *W* (the most-significant 1 to 4 bytes) is in the aligned word containing *EffAddr*. The same number of the most-significant (left) bytes from the word in GPR *rx* are stored into these bytes of *W*.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The four consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W* (2 bytes) is located in the aligned word containing the most-significant byte at 2.

3. SWL stores the most-significant 2 bytes of the low word from the source register into these 2 bytes in memory.

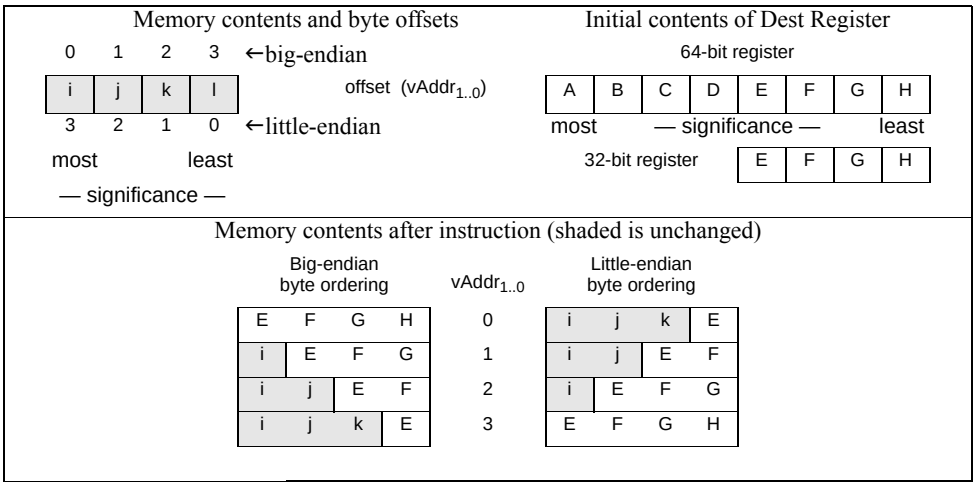
4. The complementary SWR stores the remainder of the unaligned word.

Unaligned Word Store Using SWL and SWR



The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address ($vAddr_{1..0}$)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte ordering.

Bytes Stored by an SWL Instruction



Restrictions:

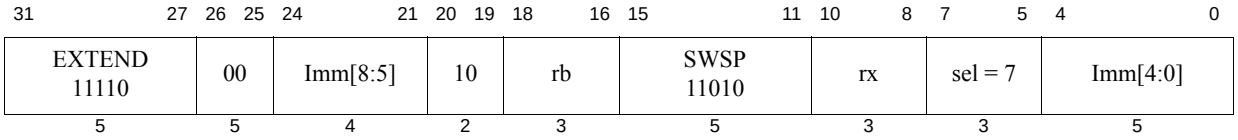
Unpredictable prior to MIPS16e2.

Operation:

```
vAddr ← sign_extend(immediate) + GPR[Xlat[rb]]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0  xor  ReverseEndian^2)
If BigEndianMem = 0 then
   pAddr ← pAddr_PSIZE-1..2 || 0^2
endif
byte ← vAddr_1..0 xor BigEndianCPU^2
dataword ← 0^24-8*byte || GPR[Xlat[rx]]_31..24-8*byte
StoreMemory(CCA, byte, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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



Format: `SWR rx, immediate(rb)` **MIPS16e2**

Purpose: Store Word Right Extended

To store the least-significant part of a word to an unaligned memory address.

Description: `memory[GPR[rb] + immediate] ← GPR[rx]`

The 9-bit signed *immediate* value is added to the contents of GPR *rb* to form an effective address *(EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

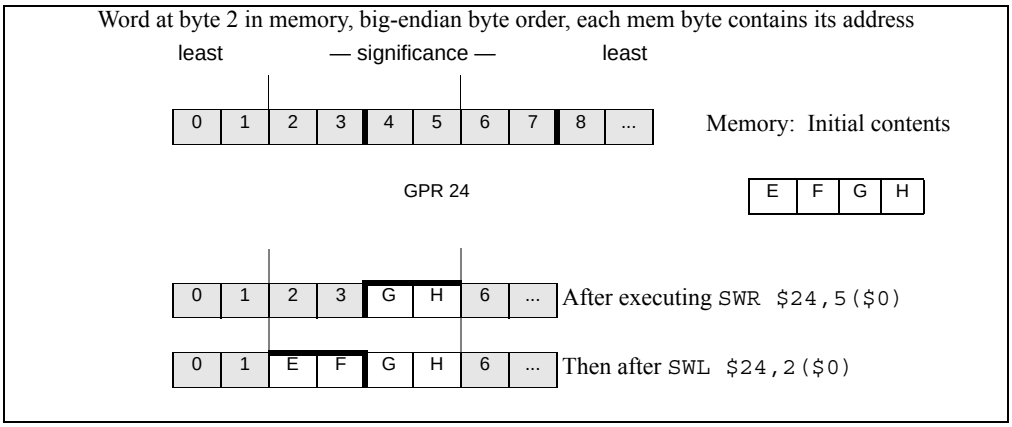
A part of *W* (the least-significant 1 to 4 bytes) is in the aligned word containing *EffAddr*. The same number of the least-significant (right) bytes from the word in GPR *rx* are stored into these bytes of *W*.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W* (2 bytes) is contained in the aligned word containing the least-significant byte at 5.

1. SWR stores the least-significant 2 bytes of the low word from the source register into these 2 bytes in memory.

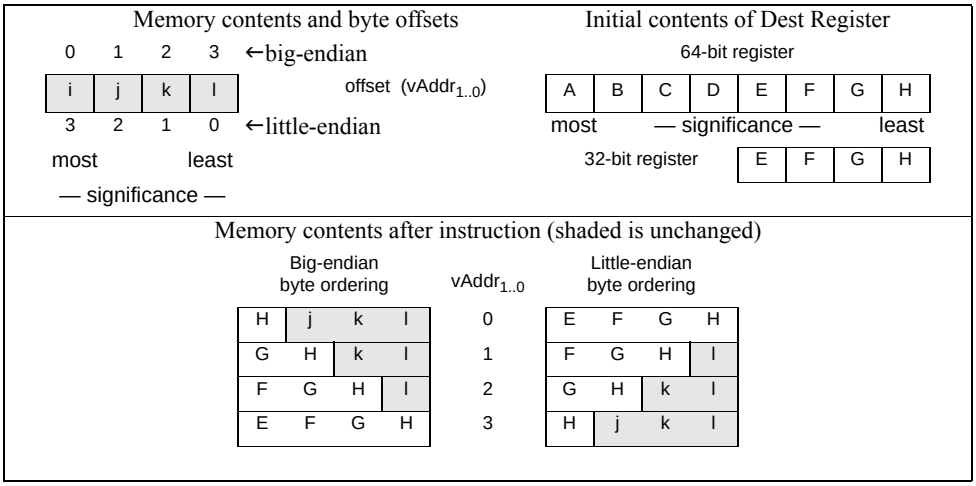
2. The complementary SWL stores the remainder of the unaligned word.

Unaligned Word Store Using SWR and SWL



The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address ($vAddr_{1..0}$)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte-ordering.

Bytes Stored by SWR Instruction



Restrictions:

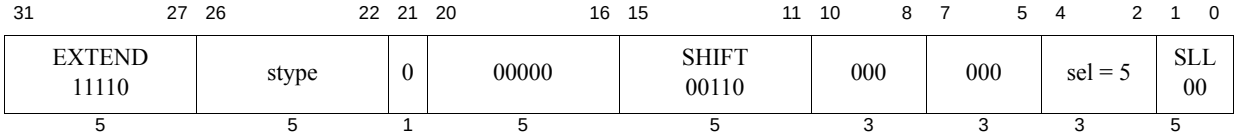
Unpredictable prior to MIPS16e2.

Operation:

```
vAddr ← sign_extend(immediate) + GPR[Xlat[rb]]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
If BigEndianMem = 0 then
   pAddr ← pAddr_PSIZE-1..2 || 0^2
endif
byte ← vAddr_1..0 xor BigEndianCPU^2
dataword ← GPR[XLat[rx]]_31-8*byte || 0^8*byte
StoreMemory(CCA, WORD-byte, dataword, pAddr, vAddr, DATA)
```

Exceptions:

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



Format: `SYNC stype` **MIPS16e2**

Purpose: Synchronize Shared Memory Extended

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 EHB, to guarantee that memory reference results are visible across operating mode changes. For example, a completion barrier is required on some implementations on entry to and exit from Debug Mode to guarantee that memory effects are handled correctly.

SYNC behavior when the stype field is zero:

- A completion barrier that affects preceding loads and stores and subsequent loads and stores.

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.

The following table lists the available completion barrier and ordering barriers behaviors that can be specified using the stype field.

Code	Name	Older instructions which must reach the load/store ordering point before the SYNC instruction completes.	Younger instructions which must reach the load/store ordering point only after the SYNC instruction completes.	Older instructions which must be globally performed when the SYNC instruction completes	Compliance
0x0	SYNC or SYNC 0	Loads, Stores	Loads, Stores	Loads, Stores	Required
0x4	SYNC_WMB or SYNC 4	Stores	Stores		Optional
0x10	SYNC_MB or SYNC 16	Loads, Stores	Loads, Stores		Optional
0x11	SYNC_ACQUIRE or SYNC 17	Loads	Loads, Stores		Optional
0x12	SYNC_RELEASE or SYNC 18	Loads, Stores	Stores		Optional
0x13	SYNC_RMB or SYNC 19	Loads	Loads		Optional
0x1-0x3, 0x5-0xF					Implementation-Specific and Vendor Specific Sync Types
0x14 - 0x1F	RESERVED				Reserved for MIPS Technologies for future extension of the architecture.

Terms:

Synchronizable: A load or store instruction is *synchronizable* if the load or store occurs to a physical location in shared memory using a virtual location with a memory access type of either *uncached* or *cached coherent*. *Shared memory* is memory that can be accessed by more than one processor or by a coherent I/O system module.

Performed load: A load instruction is *performed* when the value returned by the load has been determined. The result of a load on processor A has been *determined* with respect to processor or coherent I/O module B when a subsequent store to the location by B cannot affect the value returned by the load. The store by B must use the same memory access type as the load.

Performed store: A store instruction is *performed* when the store is observable. A store on processor A is *observable* with respect to processor or coherent I/O module B when a subsequent load of the location by B returns the value written by the store. The load by B must use the same memory access type as the store.

Globally performed load: A load instruction is *globally performed* when it is performed with respect to all processors and coherent I/O modules capable of storing to the location.

Globally performed store: A store instruction is *globally performed* when it is globally observable. It is *globally observable* when it is observable by all processors and I/O modules capable of loading from the location.

Coherent I/O module: A *coherent I/O module* is an Input/Output system component that performs coherent Direct Memory Access (DMA). It reads and writes memory independently as though it were a processor doing loads and stores to locations with a memory access type of *cached coherent*.

Load/Store Datapath: The portion of the processor which handles the load/store data requests coming from the processor pipeline and processes those requests within the cache and memory system hierarchy.

Restrictions:

Unpredictable prior to MIPS16e2. The effect of SYNC on the global order of loads and stores for memory access types other than *uncached* and *cached coherent* is **UNPREDICTABLE**.

Operation:

```
SyncOperation(stype)
```

Exceptions:

None

Programming Notes:

A processor executing load and store instructions observes the order in which loads and stores using the same memory access type occur in the instruction stream; this is known as *program order*.

A *parallel program* has multiple instruction streams that can execute simultaneously on different processors. In multiprocessor (MP) systems, the order in which the effects of loads and stores are observed by other processors—the *global order* of the loads and store—determines the actions necessary to reliably share data in parallel programs.

When all processors observe the effects of loads and stores in program order, the system is *strongly ordered*. On such systems, parallel programs can reliably share data without explicit actions in the programs. For such a system, SYNC has the same effect as a NOP. Executing SYNC on such a system is not necessary, but neither is it an error.

If a multiprocessor system is not strongly ordered, the effects of load and store instructions executed by one processor may be observed out of program order by other processors. On such systems, parallel programs must take explicit actions to reliably share data. At critical points in the program, the effects of loads and stores from an instruction stream must occur in the same order for all processors. SYNC separates the loads and stores executed on the processor into two groups, and the effect of all loads and stores in one group is seen by all processors before the effect of any load or store in the subsequent group. In effect, SYNC causes the system to be strongly ordered for the executing processor at the instant that the SYNC is executed.

Many MIPS-based multiprocessor systems are strongly ordered or have a mode in which they operate as strongly ordered for at least one memory access type. The MIPS architecture also permits implementation of MP systems that are not strongly ordered; SYNC enables the reliable use of shared memory on such systems. A parallel program that does not use SYNC generally does not operate on a system that is not strongly ordered. However, a program that does use SYNC works on both types of systems. (System-specific documentation describes the actions needed to reliably share data in parallel programs for that system.)

The behavior of a load or store using one memory access type is **UNPREDICTABLE** if a load or store was previously made to the same physical location using a different memory access type. The presence of a SYNC between the references does not alter this behavior.

SYNC affects the order in which the effects of load and store instructions appear to all processors; it does not generally affect the physical memory-system ordering or synchronization issues that arise in system programming. The effect of SYNC on implementation-specific aspects of the cached memory system, such as writeback buffers, is not

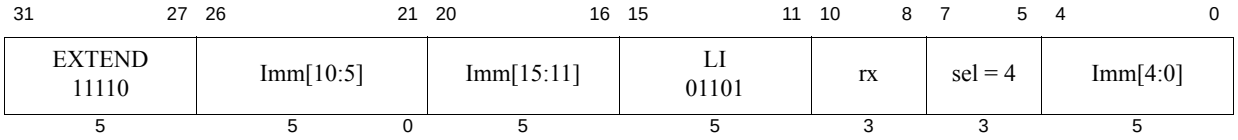
defined.

```
# Processor A (writer)
# Conditions at entry:
# The value 0 has been stored in FLAG and that value is observable by B
SW     R1, DATA         # change shared DATA value
LI     R2, 1
SYNC                    # Perform DATA store before performing FLAG store
SW     R2, FLAG         # say that the shared DATA value is valid

   # Processor B (reader)
      LI     R2, 1
   1: LW     R1, FLAG  # Get FLAG
      BNE    R2, R1, 1B# if it says that DATA is not valid, poll again
      NOP
      SYNC             # FLAG value checked before doing DATA read
      LW     R1, DATA  # Read (valid) shared DATA value
```

The code fragments above shows how SYNC can be used to coordinate the use of shared data between separate writer and reader instruction streams in a multiprocessor environment. The FLAG location is used by the instruction streams to determine whether the shared data item DATA is valid. The SYNC executed by processor A forces the store of DATA to be performed globally before the store to FLAG is performed. The SYNC executed by processor B ensures that DATA is not read until after the FLAG value indicates that the shared data is valid.

Software written to use a SYNC instruction with a non-zero stype value, expecting one type of barrier behavior, should only be run on hardware that actually implements the expected barrier behavior for that non-zero stype value or on hardware which implements a superset of the behavior expected by the software for that stype value. If the hardware does not perform the barrier behavior expected by the software, the system may fail.



Format: `XORI rx, immediate` **MIPS16e2**

Purpose: Exclusive OR Immediate Extended

To do a bitwise logical Exclusive OR with a constant.

Description: `GPR[rx] ← GPR[rx] XOR immediate`

Combine the contents of GPR *rx* and the 16-bit zero-extended *immediate* in a bitwise logical Exclusive OR operation and place the result back into GPR *rx*.

Restrictions:

Unpredictable prior to MIPS16e2.

Operation:

```
GPR[XLat[rx]] ← GPR[Xlat[rx]] xor zero_extend(immediate)
```

Exceptions:

None