MIPS64® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS64® Architecture

Document Number: MD00077
Revision 2.50
July 1, 2005

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Copyright © 2005 MIPS Technologies, Inc. All rights reserved.

Unpublished rights (if any) reserved under the copyright laws of the United States of America and other countries.

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

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

MIPS Technologies reserves the right to change the information contained in this document to improve function, design or otherwise. MIPS Technologies does not assume any liability arising out of the application or use of this information, or of any error or omission in such information. Any warranties, whether express, statutory, implied or otherwise, including but not limited to the implied warranties of merchantability or fitness for a particular purpose, are excluded. Except as expressly provided in any written license agreement from MIPS Technologies or an authorized third party, the furnishing of this document does not give recipient any license to any intellectual property rights, including any patent rights, that cover the information in this document.

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

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

MIPS, MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPS-3D, MIPS16, MIPS16e, MIPS32, MIPS64, MIPS-Based, MIPSsim, MIPSpro, MIPS Technologies logo, MIPS RISC CERTIFIED POWER logo, 4K, 4Kc, 4Km, 4Kp, 4KE, 4KEc, 4KEm, 4KEp, 4KS, 4KSc, 4KSd, M4K, 5K, 5Kc, 5Kf, 20Kc, 24K, 24Kc, 24Kf, 24KE, 24KEc, 24KEf, 25Kf, 34K, R3000, R4000, R5000, ASMACRO, Atlas, "At the core of the user experience.", BusBridge, CorExtend, CoreFPGA, CoreLV, EC, FastMIPS, JALGO, Malta, MDMX, MGB, PDtrace, the Pipeline, Pro Series, QuickMIPS, SEAD, SEAD-2, SmartMIPS, SOC-it, and YAMON are trademarks or registered trademarks of MIPS Technologies, Inc. in the United States and other countries.

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

Template: B1.14, Built with tags: 2B ARCH FPU_PS FPU_PSandARCH MIPS64

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Table of Contents

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

List of Figures

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

List of Tables

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

About This Book

The MIPS64® Architecture for Programmers Volume IV-a comes as a multi-volume set.

- Volume I describes conventions used throughout the document set, and provides an introduction to the MIPS64® Architecture
- Volume II provides detailed descriptions of each instruction in the MIPS64® instruction set
- Volume III describes the MIPS64® Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS64® processor implementation
- Volume IV-a describes the MIPS16e™ Application-Specific Extension to the MIPS64® Architecture
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64® Architecture
- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS64® Architecture
- Volume IV-d describes the SmartMIPS®Application-Specific Extension to the MIPS32® Architecture and is not applicable to the MIPS64® document set

1.1 Typographical Conventions

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

1.1.1 Italic Text

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

1.1.2 Bold Text

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

1.1.3 Courier Text

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

1.2 UNPREDICTABLE and UNDEFINED

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

1.2.1 UNPREDICTABLE

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

UNPREDICTABLE results or operations have several implementation restrictions:

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

1.2.2 UNDEFINED

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

UNDEFINED operations or behavior has one implementation restriction:

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

1.2.3 UNSTABLE

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

UNSTABLE values have one implementation restriction:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

1.3 Special Symbols in Pseudocode Notation

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

Table 1-1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
←	Assignment
=, ≠	Tests for equality and inequality
||	Bit string concatenation
x^y	A *y*-bit string formed by *y* copies of the single-bit value *x*
b#n	A constant value *n* in base *b*. For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10.
0bn	A constant value *n* in base *2*. For instance 0b100 represents the binary value 100 (decimal 4).
0xn	A constant value *n* in base *16*. For instance 0x100 represents the hexadecimal value 100 (decimal 256).
$x_{y..z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation (rightmost bit is 0) is used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
+, −	2's complement or floating point arithmetic: addition, subtraction
*, ×	2's complement or floating point multiplication (both used for either)
div	2's complement integer division
mod	2's complement modulo
/	Floating point division
<	2's complement less-than comparison
>	2's complement greater-than comparison
≤	2's complement less-than or equal comparison
≥	2's complement greater-than or equal comparison
nor	Bitwise logical NOR
xor	Bitwise logical XOR
and	Bitwise logical AND
or	Bitwise logical OR
GPRLEN	The length in bits (32 or 64) of the CPU general-purpose registers
GPR[x]	CPU general-purpose register *x*. The content of *GPR[0]* is always zero. In Release 2 of the Architecture, GPR[x] is a short-hand notation for *SGPR[SRSCtl*$_{CSS}$*, x]*.
SGPR[s,x]	In Release 2 of the Architecture, multiple copies of the CPU general-purpose registers may be implemented. *SGPR[s,x]* refers to GPR set *s*, register *x*.
FPR[x]	Floating Point operand register *x*
FCC[CC]	Floating Point condition code CC. *FCC[0]* has the same value as *COC[1]*.
FPR[x]	Floating Point (Coprocessor unit 1), general register *x*

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Table 1-1 Symbols Used in Instruction Operation Statements

<table>
<tr><th>Symbol</th><th>Meaning</th></tr>
<tr><td>CPR[z,x,s]</td><td>Coprocessor unit z, general register x, select s</td></tr>
<tr><td>CP2CPR[x]</td><td>Coprocessor unit 2, general register x</td></tr>
<tr><td>CCR[z,x]</td><td>Coprocessor unit z, control register x</td></tr>
<tr><td>CP2CCR[x]</td><td>Coprocessor unit 2, control register x</td></tr>
<tr><td>COC[z]</td><td>Coprocessor unit z condition signal</td></tr>
<tr><td>Xlat[x]</td><td>Translation of the MIPS16e GPR number x into the corresponding 32-bit GPR number</td></tr>
<tr><td>BigEndianMem</td><td>Endian mode as configured at chip reset (0 →Little-Endian, 1 → Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution.</td></tr>
<tr><td>BigEndianCPU</td><td>The endianness for load and store instructions (0 → Little-Endian, 1 → Big-Endian). In User mode, this endianness may be switched by setting the RE bit in the Status register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).</td></tr>
<tr><td>ReverseEndian</td><td>Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the RE bit of the Status register. Thus, ReverseEndian may be computed as (SR_{RE} and User mode).</td></tr>
<tr><td>LLbit</td><td>Bit of virtual state used to specify operation for instructions that provide atomic read-modify-write. LLbit is set when a linked load occurs and is tested by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.</td></tr>
<tr><td>I:,
I+n:,
I-n:</td><td>This occurs as a prefix to Operation description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to “execute.” Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of I. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction I, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled I+1.

The effect of pseudocode statements for the current instruction labelled I+1 appears to occur “at the same time” as the effect of pseudocode statements labeled I for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur “at the same time,” there is no defined order. Programs must not depend on a particular order of evaluation between such sections.</td></tr>
<tr><td>PC</td><td>The Program Counter value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to PC during an instruction time. If no value is assigned to PC during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the PC during the instruction time of the instruction in the branch delay slot.

In the MIPS Architecture, the PC value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. The PC value contains a full 64-bit address all of which are significant during a memory reference.</td></tr>
<tr><td>ISA Mode</td><td>In processors that implement the MIPS16e Application Specific Extension, the ISA Mode is a single-bit register that determines in which mode the processor is executing, as follows:

<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>The processor is executing 32-bit MIPS instructions</td></tr><tr><td>1</td><td>The processor is executing MIIPS16e instructions</td></tr></table>
In the MIPS Architecture, the ISA Mode value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the ISA Mode into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception.</td></tr>
</table>

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Table 1-1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
PABITS	The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes.
SEGBITS	The number of virtual address bits implemented in a segment of the address space is represented by the symbol SEGBITS. As such, if 40 virtual address bits are implemented in a segment, the size of the segment is $2^{SEGBITS} = 2^{40}$ bytes.
FP32RegistersMode	Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR. In MIPS32 implementations, **FP32RegistersMode** is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case **FP32RegisterMode** is computed from the FR bit in the *Status* register. If this bit is a 0, the processor operates as if it had 32 32-bit FPRs. If this bit is a 1, the processor operates with 32 64-bit FPRs. The value of **FP32RegistersMode** is computed from the FR bit in the *Status* register.
InstructionInBranchDelaySlot	Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the *dynamic* state of the instruction, not the *static* state. That is, the value is false if a branch or jump occurs to an instruction whose PC immediately follows a branch or jump, but which is not executed in the delay slot of a branch or jump.
SignalException(exception, argument)	Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument). Control does not return from this pseudocode function - the exception is signaled at the point of the call.

1.4 For More Information

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

http://www.mips.com

Comments or questions on the MIPS64® Architecture or this document should be directed to

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

or via E-mail to architecture@mips.com.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Chapter 2

Guide to the Instruction Set

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

2.1 Understanding the Instruction Fields

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

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

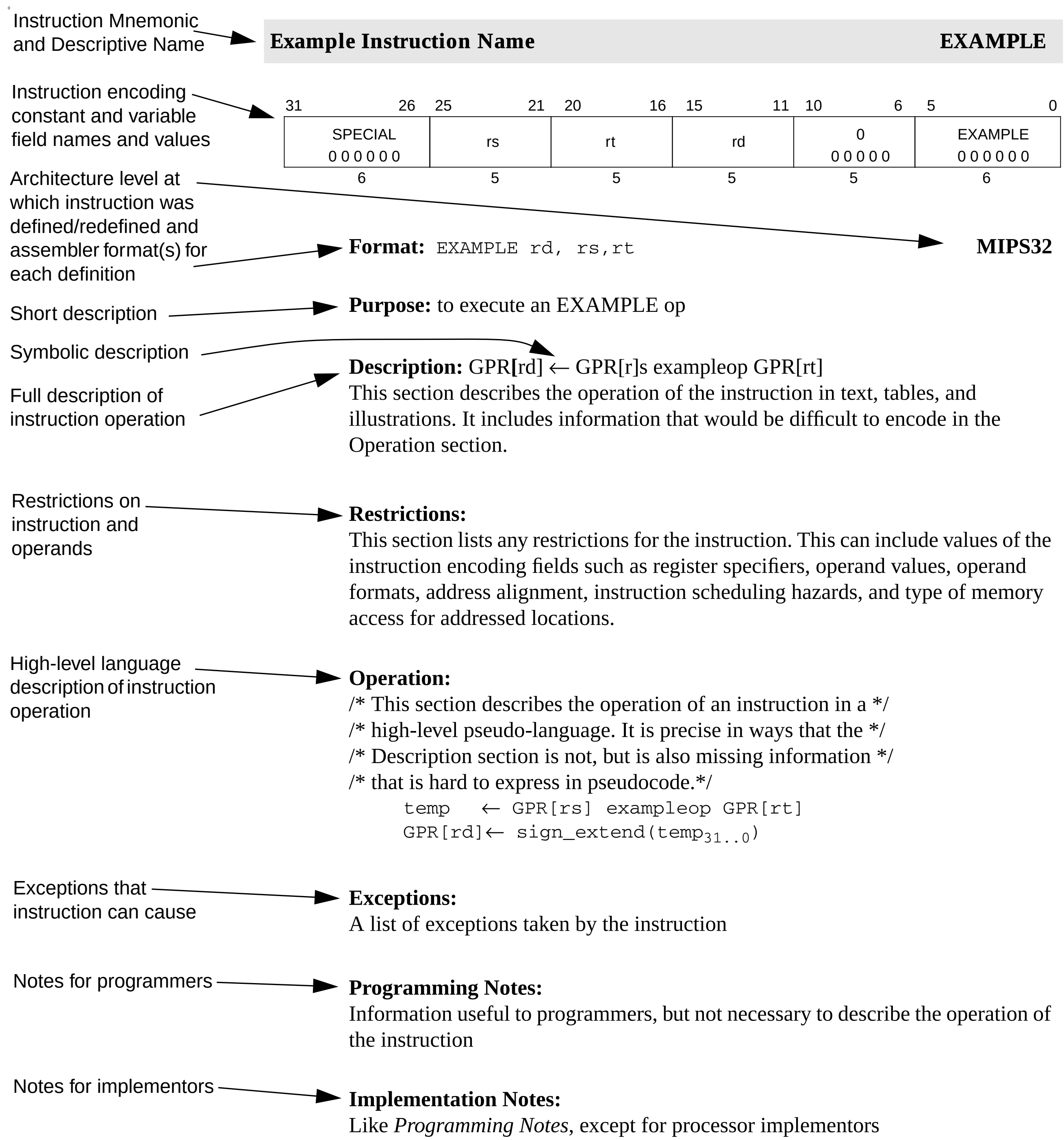

Figure 2-1 Example of Instruction Description

2.1.1 Instruction Fields

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

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

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

Figure 2-2 Example of Instruction Fields

2.1.2 Instruction Descriptive Name and Mnemonic

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

Add Word	**ADD**

Figure 2-3 Example of Instruction Descriptive Name and Mnemonic

2.1.3 Format Field

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

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

Figure 2-4 Example of Instruction Format

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

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

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

2.1.4 Purpose Field

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

Purpose:

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

Figure 2-5 Example of Instruction Purpose

2.1.5 Description Field

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

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

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

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

Figure 2-6 Example of Instruction Description

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

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

2.1.6 Restrictions Field

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

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Restrictions:

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

Figure 2-7 Example of Instruction Restrictions

2.1.7 Operation Field

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

Operation:

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

Figure 2-8 Example of Instruction Operation

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

2.1.8 Exceptions Field

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

Exceptions:

Integer Overflow

Figure 2-9 Example of Instruction Exception

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

2.1.9 Programming Notes and Implementation Notes Fields

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

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

Programming Notes:

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

Figure 2-10 Example of Instruction Programming Notes

2.2 Operation Section Notation and Functions

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

This section presents information about the following topics:

- "Instruction Execution Ordering" on page 12
- "Pseudocode Functions" on page 12

2.2.1 Instruction Execution Ordering

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

2.2.2 Pseudocode Functions

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

- "Coprocessor General Register Access Functions" on page 12
- "Memory Operation Functions" on page 14
- "Floating Point Functions" on page 17
- "Miscellaneous Functions" on page 20

2.2.2.1 Coprocessor General Register Access Functions

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

COP_LW

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

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

   /* Coprocessor-dependent action */

endfunction COP_LW
```

Figure 2-11 COP_LW Pseudocode Function

COP_LD

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

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

   /* Coprocessor-dependent action */

endfunction COP_LD
```

Figure 2-12 COP_LD Pseudocode Function

COP_SW

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

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

   /* Coprocessor-dependent action */

endfunction COP_SW
```

Figure 2-13 COP_SW Pseudocode Function

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

COP_SD

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

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

   /* Coprocessor-dependent action */

endfunction COP_SD
```

Figure 2-14 COP_SD Pseudocode Function

CoprocessorOperation

The CoprocessorOperation function performs the specified Coprocessor operation.

```
CoprocessorOperation (z, cop_fun)

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

   /* Transmit the cop_fun value to coprocessor z */

endfunction CoprocessorOperation
```

Figure 2-15 CoprocessorOperation Pseudocode Function

2.2.2.2 Memory Operation Functions

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

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

AddressTranslation

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

Given the virtual address *vAddr*, and whether the reference is to Instructions or Data (*IorD*), find the corresponding physical address (*pAddr*) and the cache coherence algorithm (*CCA*) used to resolve the reference. If the virtual address is in one of the unmapped address spaces, the physical address and *CCA* are determined directly by the virtual address. If the virtual address is in one of the mapped address spaces then the TLB or fixed mapping MMU determines the

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

physical address and access type; if the required translation is not present in the TLB or the desired access is not permitted, the function fails and an exception is taken.

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

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

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

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

endfunction AddressTranslation
```

Figure 2-16 AddressTranslation Pseudocode Function

LoadMemory

The LoadMemory function loads a value from memory.

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

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

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

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

endfunction LoadMemory
```

Figure 2-17 LoadMemory Pseudocode Function

StoreMemory

The StoreMemory function stores a value to memory.

The specified data is stored into the physical location *pAddr* using the memory hierarchy (data caches and main memory) as specified by the Cache Coherence Algorithm (*CCA*). The *MemElem* contains the data for an aligned, fixed-width memory element (a word for 32-bit processors, a doubleword for 64-bit processors), though only the bytes that are

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

actually stored to memory need be valid. The low-order two (or three) bits of *pAddr* and the *AccessLength* field indicate which of the bytes within the *MemElem* data should be stored; only these bytes in memory will actually be changed.

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

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

endfunction StoreMemory
```

Figure 2-18 StoreMemory Pseudocode Function

Prefetch

The Prefetch function prefetches data from memory.

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

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

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

endfunction Prefetch
```

Figure 2-19 Prefetch Pseudocode Function

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

Table 2-1 AccessLength Specifications for Loads/Stores

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

SyncOperation

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

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

```
SyncOperation(stype)

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

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

endfunction SyncOperation
```

Figure 2-20 SyncOperation Pseudocode Function

2.2.2.3 Floating Point Functions

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

ValueFPR

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

```
value ← ValueFPR(fpr, fmt)

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

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

    case fmt of
        S, W, UNINTERPRETED_WORD:
            valueFPR ← UNPREDICTABLE^32 || FPR[fpr]_31..0

        D, UNINTERPRETED_DOUBLEWORD:
            if (FP32RegistersMode = 0)
                if (fpr_0 ≠ 0) then
                    valueFPR ← UNPREDICTABLE
                else
                    valueFPR ← FPR[fpr+1]_31..0 || FPR[fpr]_31..0
                endif
            else
                valueFPR ← FPR[fpr]
            endif

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

        DEFAULT:
            valueFPR ← UNPREDICTABLE

    endcase
endfunction ValueFPR
```

Figure 2-21 ValueFPR Pseudocode Function

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

StoreFPR

```
StoreFPR (fpr, fmt, value)

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

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


    case fmt of
        S, W, UNINTERPRETED_WORD:
            FPR[fpr] ← UNPREDICTABLE^32 || value_31..0

        D, UNINTERPRETED_DOUBLEWORD:
            if (FP32RegistersMode = 0)
                if (fpr_0 ≠ 0) then
                    UNPREDICTABLE
                else
                    FPR[fpr]   ← UNPREDICTABLE^32 || value_31..0
                    FPR[fpr+1] ← UNPREDICTABLE^32 || value_63..32
                endif
            else
                FPR[fpr] ← value
            endif

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

    endcase

endfunction StoreFPR
```

Figure 2-22 StoreFPR Pseudocode Function

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

CheckFPException

```
CheckFPException()

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

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

endfunction CheckFPException
```

Figure 2-23 CheckFPException Pseudocode Function

FPConditionCode

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

```
tf ←FPConditionCode(cc)

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

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

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

endfunction FPConditionCode
```

Figure 2-24 FPConditionCode Pseudocode Function

SetFPConditionCode

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

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

endfunction SetFPConditionCode
```

Figure 2-25 SetFPConditionCode Pseudocode Function

2.2.2.4 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

SignalException

The SignalException function signals an exception condition.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

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

```
SignalException(Exception, argument)

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

endfunction SignalException
```

Figure 2-26 SignalException Pseudocode Function

SignalDebugBreakpointException

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

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

```
SignalDebugBreakpointException()

endfunction SignalDebugBreakpointException
```

Figure 2-27 SignalDebugBreakpointException Pseudocode Function

SignalDebugModeBreakpointException

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

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

```
SignalDebugModeBreakpointException()

endfunction SignalDebugModeBreakpointException
```

Figure 2-28 SignalDebugModeBreakpointException Pseudocode Function

NullifyCurrentInstruction

The NullifyCurrentInstruction function nullifies the current instruction.

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

```
NullifyCurrentInstruction()

endfunction NullifyCurrentInstruction
```

Figure 2-29 NullifyCurrentInstruction PseudoCode Function

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

JumpDelaySlot

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

```
JumpDelaySlot(vAddr)

   /* vAddr:Virtual address */

endfunction JumpDelaySlot
```

Figure 2-30 JumpDelaySlot Pseudocode Function

NotWordValue

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

```
result ← NotWordValue(value)

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

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

   NotWordValue ← value63..32 ≠ (value31)^32

endfunction NotWordValue
```

Figure 2-31 NotWordValue Pseudocode Function

PolyMult

The PolyMult function multiplies two binary polynomial coefficients.

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

   PolyMult ← temp

endfunction PolyMult
```

Figure 2-32 PolyMult Pseudocode Function

2.3 Op and Function Subfield Notation

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

2.4 FPU Instructions

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

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

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

See Section 2.3, "Op and Function Subfield Notation" on page 22 for a description of the *op* and *function* subfields.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

The MIPS16e™ Application-Specific Extension to the MIPS64® Architecture

This chapter describes the purpose and key features of the MIPS16e™ Application-Specific Extension (ASE) to the MIPS64® Architecture. The MIPS16e ASE is an enhancement to the previous MIPS16™ ASE which provides additional instructions to improve the compaction of the code.

3.1 Base Architecture Requirements

The MIPS16e ASE requires the following base architecture support:

- **The MIPS32 or MIPS64 Architecture**: The MIPS16e ASE requires a compliant implementation of the MIPS32 or MIPS64 Architecture.

3.2 Software Detection of the ASE

Software may determine if the MIPS16e ASE is implemented by checking the state of the CA bit in the *Config1* CP0 register.

3.3 Compliance and Subsetting

There are no instruction subsets of the MIPS16e ASE to the MIPS64 Architecture — all MIPS16e instructions must be implemented. Specifically, this means that the original MIPS16 ASE is not an allowable subset of the MIPS16e ASE. For the MIPS16e 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.

3.4 MIPS16e Overview

The MIPS16e ASE allows embedded designs to substantially reduce system cost by reducing overall memory requirements. The MIPS16e 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 MIPS16e ASE.

The MIPS16e 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 MIPS16e 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.

This chapter presents specific information about the following topics:

- "MIPS16e ASE Features" on page 26
- "MIPS16e Register Set" on page 26

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

3.5 MIPS16e ASE Features

The MIPS16e ASE includes the following features:

- allows MIPS16e instructions to be intermixed with existing MIPS instruction binaries
- is compatible with the MIPS32 and MIPS64 instruction sets
- allows switching between MIPS16e 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 MIPS16e 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.

3.6 MIPS16e Register Set

The MIPS16e register set is listed in Table 3-1 and Table 3-2. This register set is a true subset of the register set available in 32-bit mode; the MIPS16e ASE can directly access 8 of the 32 registers available in 32-bit mode.

In addition to the eight general-purpose registers, 0-7, listed in Table 3-1, specific instructions in the MIPS16e ASE reference the stack pointer register (*sp*), the return address register (*ra*), the condition code register (*t8*), and the program counter (*PC*). Of these, Table 3-1 lists *sp*, *ra*, and *t8*, and Table 3-2 lists the MIPS16e special-purpose registers, including *PC*.

The MIPS16e ASE also contains two move instructions that provide access to all 32 general-purpose registers.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Table 3-1 MIPS16e 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
N/A	24	t8	MIPS16e *Condition Code* register; implicitly referenced by the BTEQZ, BTNEZ, CMP, CMPI, SLT, SLTU, SLTI, and SLTIU instructions
N/A	29	sp	Stack pointer register
N/A	31	ra	Return address register

1. “0-7” correspond to the register’s MIPS16e binary encoding and show how that encoding relates to the MIPS registers. “0-7” never refer to the registers, except within the binary MIPS16e 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 MIPS16e binary encoding for this register is 001.
2. General registers not shown in the above table are not accessible through the MIPS16e instruction set, except by using the Move instructions. The MIPS16e Move instructions can access all 32 general-purpose registers.
3. The MIPS16e 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 3-2 MIPS16e 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.

3.7 MIPS16e ISA Modes

This section describes the following:

- the ISA modes available in the architecture, page 28
- the purpose of the *ISA Mode* field, page 28

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

- how to switch between 32-bit MIPS and MIPS16e modes, page 28
- the role of the jump instructions when switching modes, page 28

3.7.1 Modes Available in the MIPS16e Architecture

There are two ISA modes defined in the MIPS16e Architecture, as follows:

- MIPS 32-bit mode (32-bit instructions)
- MIPS16e mode (16-bit instructions)

3.7.2 Defining the ISA Mode Field

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

Table 3-3 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.

In MIPS 32-bit mode and MIPS16e mode, the JALX, JR, JALR, JALRC, and JRC instructions can change the *ISA Mode* bit, as described in Section 3.7.4, "Using MIPS16e Jump Instructions to Switch Modes".

3.7.3 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.

3.7.4 Using MIPS16e Jump Instructions to Switch Modes

The MIPS16e application-specific extension supports procedure calls and returns from both MIPS16e and 32-bit MIPS code to both MIPS16e and 32-bit MIPS code. The following instructions are used:

- The JAL instruction supports calls to the same ISA.
- The JALX instruction supports calls that change the ISA.
- The JALR and JALRC instructions support calls to either ISA.
- The JR and JRC instructions support returns to either ISA.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

The JAL, JALR, JALRC, and JALX instructions save the *ISA Mode* bit in bit 0 of the general register containing the return address. The contents of this general register may be used by a future JR, JRC, JALR, or JALRC instruction to return and restore the ISA Mode.

The JALX instruction in both modes switches to the other ISA (it changes 0b0 → 0b1 and 0b1 → 0b0).

The JR and JALR instructions in both modes load the *ISA Mode* bit from bit 0 of the general register holding the target address. Bit 0 of the general register is not part of the target address; bit 0 of PC is loaded with a 0 so that no address exceptions can occur.

The JRC and JALRC instructions in MIPS16e mode load the *ISA Mode* bit from bit 0 of the general register holding the target address. Bit 0 of the general register is not part of the target address; bit 0 of PC is loaded with a 0 so that no address exceptions can occur.

3.8 JALX, JR, and JALR Operations in MIPS16e and MIPS32 Mode

The behavior of three of the 32-bit MIPS instructions—JALX, JR, JALR—differs between those processors that implement MIPS16e and those processors that do not.

In processors that implement the MIPS16e ASE, the three instructions behave as follows:

- The JALX instruction executes a JAL and switches to the other mode.
- JR and JALR instructions load the *ISA Mode* bit from bit 0 of the source register. Bit 0 of PC is loaded with a 0, and no Address exception can occur when bit 0 of the source register is a 1 (MIPS16e mode).

In CPUs that do not implement the MIPS16e ASE, the three instructions behave as follows:

- JALX instructions cause a Reserved Instruction exception.
- JR or JALR instructions cause an Address exception on the target instruction fetch when bit 0 of the source register is a 1.

3.9 MIPS16e Instruction Summaries

This section describes the various instruction categories and then summarizes the MIPS16e 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.
- **Save and Restore**—These instructions create and tear down stack frames.
- **Computational**—These instructions perform arithmetic, logical, and shift operations on values in registers.
- **Jump and Branch**—These instructions change the control flow of a program.
- **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.
- **Implemention-Definable Macro Instructions**—This category includes the capability of defining macros that are replaced at execution time by a set of 32-bit MIPS instructions, with appropriate parameter substitution.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Tables 3-4 through 3-12 list the MIPS16e instruction set.

Table 3-4 MIPS16e Load and Store Instructions

Mnemonic	Instruction	Extensible Instruction?	Implemented Only on MIPS64 Processors?
LB	Load Byte	Yes	No
LBU	Load Byte Unsigned	Yes	No
LD	Load Doubleword	Yes	Yes
LH	Load Halfword	Yes	No
LHU	Load Halfword Unsigned	Yes	No
LW	Load Word	Yes	No
LWU	Load Word Unsigned	Yes	Yes
SB	Store Byte	Yes	No
SD	Store Doubleword	Yes	Yes
SH	Store Halfword	Yes	No
SW	Store Word	Yes	No

Table 3-5 MIPS16e Save and Restore Instructions

Mnemonic	Instruction	Extensible Instruction?	Implemented Only on MIPS64 Processors?
RESTORE	Restore Registers and Deallocate Stack Frame	Yes	No
SAVE	Save Registers and SetUp Stack Frame	Yes	No

Table 3-6 MIPS16e ALU Immediate Instructions

Mnemonic	Instruction	Extensible Instruction?	Implemented Only on MIPS64 Processors?
ADDIU	Add Immediate Unsigned	Yes	No
CMPI	Compare Immediate	Yes	No
DADDIU	Doubleword Add Immediate Unsigned	Yes	Yes
LI	Load Immediate	Yes	No
SLTI	Set on Less Than Immediate	Yes	No
SLTIU	Set on Less Than Immediate Unsigned	Yes	No

Table 3-7 MIPS16e Arithmetic One, Two or Three Operand Register Instructions

Mnemonic	Instruction	Extensible Instruction?	Implemented Only on MIPS64 Processors?
ADD	Add Unsigned	No	No
AND	AND	No	No
CMP	Compare	No	No
DADDU	Doubleword Add Unsigned	No	Yes
DSUBU	Doubleword Subtract Unsigned	No	Yes
MOVE	Move	No	No
NEG	Negate	No	No
NOT	Not	No	No
OR	OR	No	No
SEB	Sign-Extend Byte	No	No

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Mnemonic	Instruction	Extensible Instruction?	Implemented Only on MIPS64 Processors?
SEH	Sign-Extend Halfword	No	No
SEW	Sign-Extend Word	No	Yes
SLT	Set on Less Than	No	No
SLTU	Set on Less Than Unsigned	No	No
SUBU	Subtract Unsigned	No	No
XOR	Exclusive OR	No	No
ZEB	Zero-extend Byte	No	No
ZEH	Zero-Extend Halfword	No	No
ZEW	Zero-Extend Word	No	Yes

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Table 3-8 MIPS16e Special Instructions

Mnemonic	Instruction	Extensible Instruction?	Implemented Only on MIPS64 Processors?
BREAK	Breakpoint	No	No
EXTEND	Extend	No	No

Table 3-9 MIPS16e Multiply and Divide Instructions

Mnemonic	Instruction	Extensible Instruction?	Implemented Only on MIPS64 Processors?
DDIV	Doubleword Divide	No	Yes
DDIVU	Doubleword Divide Unsigned	No	Yes
DIV	Divide	No	No
DIVU	Divide Unsigned	No	No
DMULT	Doubleword Multiply	No	Yes
DMULTU	Doubleword Multiply Unsigned	No	Yes
MFHI	Move From HI	No	No
MFLO	Move From LO	No	No
MULT	Multiply	No	No
MULTU	Multiply Unsigned	No	No

Table 3-10 MIPS16e Jump and Branch Instructions

Mnemonic	Instruction	Extensible Instruction?	Implemented Only on MIPS64 Processors?
B	Branch Unconditional	Yes	No
BEQZ	Branch on Equal to Zero	Yes	No
BNEZ	Branch on Not Equal to Zero	Yes	No
BTEQZ	Branch on T Equal to Zero	Yes	No
BTNEZ	Branch on T Not Equal to Zero	Yes	No
JAL[1]	Jump and Link	No	No
JALR	Jump and Link Register	No	No
JALRC	Jump and Link Register Compact	No	No
JALX[1]	Jump and Link Exchange	No	No
JR	Jump Register	No	No
JRC	Jump Register Compact	No	No

1. The JAL and JALX instructions are not extensible because they are inherently 32-bit instructions.

Table 3-11 MIPS16e Shift Instructions

Mnemonic	Instruction	Extensible Instruction?	Implemented Only on MIPS64 Processors?
DSLL	Doubleword Shift Left Logical	Yes	Yes
DSLLV	Doubleword Shift Left Logical Variable	No	Yes
DSRA	Doubleword Shift Right Arithmetic	Yes	Yes
DSRAV	Doubleword Shift Right Arithmetic Variable	No	Yes
DSRL	Doubleword Shift Right Logical	Yes	Yes
DSRLV	Doubleword Shift Right Logical Variable	No	Yes

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Mnemonic	Instruction	Extensible Instruction?	Implemented Only on MIPS64 Processors?
SRA	Shift Right Arithmetic	Yes	No
SRAV	Shift Right Arithmetic Variable	No	No
SLL	Shift Left Logical	Yes	No
SLLV	Shift Left Logical Variable	No	No
SRL	Shift Right Logical	Yes	No
SRLV	Shift Right Logical Variable	No	No

Table 3-12 Implementation-Definable Macro Instructions

Mnemonic	Instruction	Extensible Instruction?	Implemented Only on MIPS64 Processors?
ASMACRO	Implementation-Definable Macro Instructions	Yes[1]	No

1. The Implementation-Definable Macro Instructions are always extended instructions. There are no 16-bit macro instruction

3.10 MIPS16e PC-Relative Instructions

The MIPS16e ASE provides PC-relative addressing for four instructions, in both extended and non-extended versions. The four instructions are listed in Table 3-13.

Table 3-13 PC-Relative MIPS16e Instructions

Instruction	Use
Load Word	LW rx, offset(pc)
Load Doubleword	LD ry, offset(pc)
Add Immediate Unsigned	ADDIU rx, pc, immediate
Doubleword Add Immediate Unsigned	DADDIU ry, pc, immediate

These instructions use the PC value of either the PC-relative instruction itself or the PC value for the preceding instruction as the base for address calculation.

Table 3-14 summarizes the address calculation base used for the various instruction combinations.

Table 3-14 PC-Relative Base Used for Address Calculation

Instruction	BasePC Value
Non-extended PC-relative instruction not in Jump Delay Slot	Address of instruction
Extended PC-relative instruction	Address of Extend instruction
Non-extended PC-relative instruction in JR or JALR jump delay slot	Address of JR or JALR instruction
Non-extended PC-relative instruction in JAL or JALX jump delay slot	Address of first JAL or JALX halfword

The JRC and JALRC instructions do not have delay slots and do not affect the PC-relative base address calculated for an instruction immediately following the JRC or JALRC.

In the descriptive summaries of PC-relative instructions, located in Tables 3-13 and 3-14, the PC value used as the basis for calculating the address is referred to as the BasePC value. The BasePC equals the *Exception Program Counter* (*EPC*) value associated with the PC-relative instruction.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

3.11 MIPS16e Extensible Instructions

This section explains the purpose of an *Extend* instruction, how to use it, and which MIPS16e instructions are extensible.

The Extend instruction allows you to enlarge the *immediate* field of any MIPS16e instruction whose *immediate* field is smaller than the *immediate* field in the equivalent 32-bit MIPS instruction. The Extend instruction is a prefix which modifies the behavior of the instruction which follows it, and must always immediately precede the instruction whose *immediate* field you want to extend. Every extended instruction uses 4 bytes in program memory instead of 2 bytes (2 bytes for Extend and 2 bytes for the instruction being extended), and it can cross a word boundary. The PC value of an extended instruction is the address of the halfword containing the Extend.

For example, the following MIPS16e instruction contains a five-bit *immediate*.

```
LW ry, offset(rx)
```

The *immediate* expands to 16 bits (0b000000000 || offset || 0b00) before execution in the pipeline. This allows 32 different offset values of 0, 4, 8, and up through 124, in increments of 4. Once extended, this instruction can hold any of the 65,536 values in the range -32768 through 32767 that are also available with the 32-bit MIPS version of the LW instruction.

Shift instructions are extended to unsigned *immediate*s of 5 bits. All other immediate instructions expand to either signed or unsigned 16-bit immediates. There are only two exceptions which can be extended to a 15-bit signed *immediate*:

```
ADDIU ry, rx, immediate
DADDIU ry, rx, immediate
```

Unlike most other extended instructions, an extended RESTORE or SAVE instruction provides both a larger frame size adjustment, and the ability to save and restore more registers than the non-extended version.

Once both halves of an extended instruction have been fetched and the instruction starts flowing down the pipeline, the instruction is treated as a single entity, not as independent instructions. This implies that an exception or interrupt never reports an EPC value between the EXTEND and the instruction being extended, and that EJTAG single step treats an instruction step as the execution of the entire extended instruction, not the components.

There is only one restriction on the location of extensible instructions: They may not be placed in jump delay slots. Doing so causes **UNPREDICTABLE** results.

Table 3-15 lists the MIPS16e extensible instructions, the size of their *immediate*, and how much each *immediate* can be extended when preceded with an Extend instruction. Executing an instruction which is not extensible (those which are maked No in the “Extensible Instruction?” column of Table 3-4 through Table 3-12, including the EXTEND instruction itself) must cause a Reserved Instruction Exception.

Table 3-15 MIPS16e Extensible Instructions

Mnemonic	MIPS16e Instruction	MIPS16e Immediate	Extended Immediate
ADDIU	Add Immediate Unsigned	4 (ADDIU ry, rx, imm)	15 (ADDIU ry, rx, imm)
		8	16
B	Branch Unconditional	11	16
BEQZ	Branch on Equal to Zero	8	16
BNEZ	Branch on Not Equal to Zero	8	16
BTEQZ	Branch on T Equal to Zero	8	16
BTNEZ	Branch on T Not Equal to Zero	8	16
CMPI	Compare Immediate	8	16

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Mnemonic	MIPS16e Instruction	MIPS16e Immediate	Extended Immediate
DADDIU	Doubleword Add Immediate Unsigned	4 (DADDIU ry, rx, imm) 5 (or 8)	15 (DADDIU ry, rx, imm) 16
DSLL	Doubleword Shift Left Logical	3	6
DSRA	Doubleword Shift Right Arithmetic	3	6
DSRL	Doubleword Shift Right Logical	3	6
LB	Load Byte	5	16
LBU	Load Byte Unsigned	5	16
LD	Load Doubleword	5	16
LH	Load Halfword	5	16
LHU	Load Halfword Unsigned	5	16
LI	Load Immediate	8	16
LW	Load Word	5 (or 8)	16
LWU	Load Word Unsigned	5	16
RESTORE	Restore Registers and Deallocate Stack Frame	4	8
SAVE	Save Registers and Set Up Stack Frame	4	8
SB	Store Byte	5	16
SD	Store Doubleword	5 (or 8)	16
SH	Store Halfword	5	16
SLL	Shift Left Logical	3	5
SLTI	Set on Less Than Immediate	8	16
SLTIU	Set on Less Than Immediate Unsigned	8	16
SRA	Shift Right Arithmetic	3	5
SRL	Shift Right Logical	3	5
SW	Store Word	5 (or 8)	16

3.12 MIPS16e Implementation-Definable Macro Instructions

Previous revisions of the MIPS16e ASE assumed that most MIPS16e instructions mapped to a single 32-bit MIPS instruction. However, there are several MIPS16e instructions for which there is no corresponding 32-bit MIPS instruction equivalent. The addition of the SAVE and RESTORE instructions introduced the possibility that a single MIPS16e instruction expand to a fixed sequence of multiple 32-bit instructions. The obvious extension to this capability is the ability to define a *Macro* capability in which a single extended MIPS16e instruction can be expanded into a sequence of 32-bit MIPS instructions, with parameter substitution done between fields of the macro instruction and fields of the expanded instructions. This is the concept behind the addition of Implementation-Definable Macro Instructions to the MIPS16e ASE.

The term “Implementation-Definable” refers to the fact that the macro definitions are created when the processor is implemented, rather than via a programmable mechanism that is available to the user of the processor. The macro definitions, expansions, and parameter substitutions are defined when the processor is implemented, and is therefore implementation-dependent. The programmer visible representation of this macro capability is provided by the ASMACRO (for Application Specific Macro) instruction, as defined in the next chapter.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

3.13 MIPS16e Jump and Branch Instructions

Jump and Branch instructions change the control flow of a program.

The JAL, JALR, JALX, and JR instructions occur with a one-instruction delay. That is, the instruction immediately following the jump is always executed, whether or not the jump is taken.

Branch instructions and the JALRC and JRC jump instructions do not have a delay slot. If a branch or jump is taken, the instruction immediately following the branch or jump is never executed. If the branch or jump is not taken, the instruction following the branch or jump is always executed.

Branch, jump and extended instructions may not be placed in jump delay slots. Doing so causes **UNPREDICTABLE** results.

3.14 MIPS16e Instruction Formats

This section defines the format[1] for each MIPS16e instruction type and includes formats for both normal and extended instructions.

Every MIPS16e 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 3-16.

Table 3-16 MIPS16e Instruction Fields

Field	Definition
funct or f	Function field
immediate or imm	4-, 5-, 8-, or 11-bit immediate, branch displacement, or address displacement
op	5-bit major operation code
rx	3-bit source or destination register specifier
ry	3-bit source or destination register specifier
rz	3-bit source or destination register specifier
sa	3- or 5-bit shift amount

[1] As used here, the term *format* means the layout of the MIPS16e instruction word.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

3.14.1 I-type instruction format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op					immediate										

3.14.2 RI-type instruction format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op					rx			immediate							

3.14.3 RR-type instruction format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RR					rx			ry[1]			funct				

1. When the funct field is either *CNVT* or *J(AL)R(C)*, the *ry* field encodes a sub-function to be performed rather than a register number

3.14.4 RRI-type instruction format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op					rx			ry			immediate				

3.14.5 RRR-type instruction format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RRR					rx			ry			rz			f	

3.14.6 RRI-A type instruction format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RRI-A					rx			ry			f	immediate			

3.14.7 Shift instruction format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHIFT					rx			ry			sa[1]			f	

1. The three-bit *sa* field can encode a shift amount of 0 through 7. 0 bit shifts (NOPs) are not possible; a 0 value translates to a shift amount of 8.

3.14.8 I8-type instruction format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I8					funct			immediate							

3.14.9 I8_MOVR32 instruction format (used only by the MOVR32 instruction)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I8					funct			ry				r32[4:0]			

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

3.14.10 I8_MOV32R instruction format (used only by MOV32R instruction)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I8					funct			r32[2:0,4:3][1]					rz		

1. The *r32* field uses special bit encoding. For example, the encoding for $7 (00111) is 11100 in the *r32* field.

3.14.11 I8_SVRS instruction format (used only by the SAVE and RESTORE instructions)

15	14	13	12	11	10	9	8	7	6	5	4	3			0
I8					SVRS			s	ra	s0	s1	framesize			

3.14.12 I64-type instruction format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I64					funct			immediate							

3.14.13 RI64-type instruction format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I64					funct			ry			immediate				

3.14.14 JAL and JALX instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JAL					X[1]	immediate 20:16					immediate 25:21					immediate 15:0															

1. If x=0, instruction is JAL. If x=1, instruction is JALX.

3.14.15 EXT-I instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTEND					immediate 10:5						immediate 15:11					op					0	0	0	0	0	0	immediate 4:0				

3.14.16 ASMACRO instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTEND					select			p4			p3					RRR					p2			p1			p0				

3.14.17 EXT-RI instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTEND					immediate 10:5						immediate 15:11					op					rx			0	0	0	immediate 4:0				

3.14.18 EXT-RRI instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTEND					immediate 10:5						immediate 15:11					op					rx			ry			immediate 4:0				

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

3.14.19 EXT-RRI-A instruction format

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

EXTEND	immediate 10:4	imm 14:11	RRI-A	rx	ry	f	imm 3:0

3.14.20 EXT-SHIFT instruction format

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

EXTEND	sa 4:0	s5[1]	0	0	0	0	0	SHIFT	rx	ry	0	0	0	f

1. s5 is equivalent to sa5, the most significant bit of the 6-bit shift amount (*sa*) field. For extended DSLL shifts, this bit may be either 0 or 1. For all 32-bit extended shifts, s5 must be 0. None of the extended shift instructions perform the 0-to-8 mapping, so 0 bit shifts are possible using the extended format.

3.14.21 EXT-I8 instruction format

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

EXTEND	immediate 10:5	immediate 15:11	I8	funct	0	0	0	immediate 4:0

3.14.22 EXT-I8_SVRS instruction format (used only by the SAVE and RESTORE instructions)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

EXTEND	xsregs	framesize 7:4	0	aregs	I8	SVRS	s	ra	s0	s1	framesize 3:0

3.14.23 EXT-I64 instruction format

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

EXTEND	immediate 10:5	immediate 15:11	I64	funct	0	0	0	immediate 4:0

3.14.24 EXT-RI64 instruction format

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

EXTEND	immediate 10:5	immediate 15:11	I64	funct	ry	immediate 4:0

3.14.25 EXT-SHIFT64 instruction format

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

EXTEND	sa 4:0	s5[1]	0	0	0	0	0	RR	0	0	0	ry	function

1. s5 is equivalent to sa5, the most-significant bit of the 6-bit shift amount (*sa*) field. None of the extended shift instructions perform the 0-to-8 mapping, so 0 bit shifts are possible using the extended format.

3.15 Instruction Bit Encoding

Table 3-18 through Table 3-26 describe the encoding used for the MIPS16e ASE. Table 3-17 describes the meaning of the symbols used in the tables.

Table 3-17 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction must cause a Reserved Instruction Exception.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Table 3-17 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
δ	(Also *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
β	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level. Executing such an instruction must cause a Reserved Instruction Exception.
⊥	Operation or field codes marked with this symbol represent instructions which are not legal if the processor is configured to be backward compatible with MIPS32 processors. If the processor is executing in Kernel Mode, Debug Mode, or 64-bit instructions are enabled, execution proceeds normally. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
θ	Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, MIPS Technologies will assist the partner in selecting appropriate encodings if requested by the partner. The partner is not required to consult with MIPS Technologies when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (*SPECIAL2* encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
σ	Field codes marked with this symbol represent an EJTAG support instruction and implementation of this encoding is optional for each implementation. If the encoding is not implemented, executing such an instruction must cause a Reserved Instruction Exception. If the encoding is implemented, it must match the instruction encoding as shown in the table.
ε	Operation or field codes marked with this symbol are reserved for MIPS Application Specific Extensions. If the ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception.
ϕ	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS64 ISA. Software should avoid using these operation or field codes.
œ	Operation or field codes marked with this symbol are not extensible (see Section 3.11, "MIPS16e Extensible Instructions" on page 34). Executing such an instruction with an EXTEND prefix must cause a Reserved Instruction Exception.

Table 3-18 MIPS16e Encoding of the Opcode Field

opcode	bits 13..11								
		0	1	2	3	4	5	6	7
bits 15..14		000	001	010	011	100	101	110	111
0	00	ADDIUSP[1]	ADDIUPC[2]	B	*JAL(X)* δ	BEQZ	BNEZ	*SHIFT* δ	LD ⊥
1	01	*RRI-A* δ	ADDIU8[3]	SLTI	SLTIU	*I8* δ	LI	CMPI	SD ⊥
2	10	LB	LH	LWSP[4]	LW	LBU	LHU	LWPC[5]	LWU ⊥
3	11	SB	SH	SWSP[6]	SW	*RRR* δ	*RR* δ	*EXTEND* δ∉	*I64* δ⊥

1. The ADDIUSP opcode is used by the ADDIU rx, sp, immediate instruction
2. The ADDIUPC opcode is used by the ADDIU rx, pc, immediate instruction
3. The ADDIU8 opcode is used by the ADDIU rx, immediate instruction
4. The LWSP opcode is used by the LW rx, offset(sp) instruction
5. The LWPC opcode is used by the LW rx, offset(pc) instruction
6. The SWSP opcode is used by the SW rx, offset(sp) instruction

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Table 3-19 MIPS16e JAL(X) Encoding of the x Field

x	bit 26	
	0	1
	JAL ∉	JALX ∉

Table 3-20 MIPS16e SHIFT Encoding of the f Field

f	bits 1..0			
	0	1	2	3
	00	01	10	11
	SLL	DSLL ⊥	SRL	SRA

Table 3-21 MIPS16e RRI-A Encoding of the f Field

f	bit 4	
	0	1
	ADDIU[1]	DADDIU[2] ⊥

1. The ADDIU function is used by the ADDIU ry, rx, immediate instruction
2. The DADDIU function is used by the DADDIU ry, rx, immediate instruction

Table 3-22 MIPS16e I8 Encoding of the funct Field

funct	bits 10..8							
	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
	BTEQZ	BTNEZ	SWRASP[1]	ADJSP[2]	*SVRS* δ	MOV32R[3] ∉	*	MOVR32[4] ∉

1. The SWRASP function is used by the SW ra, offset(sp) instruction
2. The ADJSP function is used by the ADDIU sp, immediate instruction
3. The MOV32R function is used by the MOVE r32, rz instruction
4. The MOVR32 function is used by the MOVE ry, r32 instruction

Table 3-23 MIPS16e RRR Encoding of the f Field

f	bits 1..0			
	0	1	2	3
	00	01	10	11
	DADDU ⊥∉	ADDU ∉	DSUBU ⊥∉	SUBU ∉

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Table 3-24 MIPS16e RR Encoding of the Funct Field

funct		bits 2..0							
		0	1	2	3	4	5	6	7
bits 4..3		000	001	010	011	100	101	110	111
0	00	*J(AL)R(C)* δ	SDBBP ∉	SLT ∉	SLTU ∉	SLLV ∉	BREAK ∉	SRLV ∉	SRAV ∉
1	01	DSRL ⊥	*	CMP ∉	NEG ∉	AND ∉	OR ∉	XOR ∉	NOT ∉
2	10	MFHI ∉	*CNVT* δ	MFLO ∉	DSRA ⊥	DSLLV ⊥∉	*	DSRLV ⊥∉	DSRAV ⊥∉
3	11	MULT ∉	MULTU ∉	DIV ∉	DIVU ∉	DMULT ⊥∉	DMULTU ⊥∉	DDIV ⊥∉	DDIVU ⊥∉

Table 3-25 MIPS16e I64 Encoding of the funct Field

funct	bits 10..8							
	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
	LDSP[1] ⊥	SDSP[2] ⊥	SDRASP[3] ⊥	DADJSP[4] ⊥	LDPC[5] ⊥	DADDIU5[6] ⊥	DADDIUPC[7] ⊥	DADDIUSP[8] ⊥

1. The LDSP function is used by the LD ry, offset(sp) instruction
2. The SDSP function is used by the SD ry, offset(sp) instruction
3. The SDRASP function is used by the SD ra, offset(sp) instruction
4. The DADJSP function is used by the DADDIU sp, immediate instruction
5. The LDPC function is used by the LD ry, offset(pc) instruction
6. The DADDIU5 function is used by the DADDIU ry, immediate instruction
7. The DADDIUPC function is used by the DADDIU ry, pc, immediate instruction
8. The DADDIUSP function is used by the DADDIU ry, sp, immediate instruction

Table 3-26 MIPS16e I8 Encoding of the s Field when funct=SVRS

s	bit 7	
	0	1
	RESTORE	SAVE

Table 3-27 MIPS16e RR Encoding of the ry Field when funct=J(AL)R(C)

ry	bits 7..5							
	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
	JR rx ∉	JR ra ∉	JALR ∉		JRC rx ∉	JRC ra ∉	JALRC ∉	

Table 3-28 MIPS16e RR Encoding of the ry Field when funct=CNVT

ry	bits 7..5							
	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
	ZEB ∉	ZEH ∉	ZEW ⊥∉	*	SEB ∉	SEH ∉	SEW ⊥∉	*

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

3.16 MIPS16e 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. Similarly, JAL and JALX instructions should be viewed as consisting of two 16-bit halfwords, which means this rule also applies to them.

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.

3.17 MIPS16e Instruction Fetch Restrictions

When the processor is running in MIPS16e 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.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

The MIPS16e™ ASE Instruction Set

4.1 MIPS16e Instruction Descriptions

This chapter provides an alphabetical listing of the instructions listed in Table 3-4 through Table 3-12.

4.1.1 MIPS16e-Specific Pseudocode Functions

This section defines the pseudocode functions that are specific to the MIPS16e ASE. These functions are used in the Operation section of each MIPS16e instruction description.

4.1.1.1 Xlat

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.

```
PhyReg ← Xlat(i)

    /* PhyReg:    Physical register index, in the range 0..7 */

    /* i:         Opcode register field index */

    if (i < 2) then
        Xlat ← i + 16
    else
        Xlat ← i
    endif

endfunction Xlat
```

Figure 4-1 Xlat Pseudocode Function

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Add Immediate Unsigned Word (2-Operand) ADDIU

15 … 11	10 … 8	7 … 0
ADDIU8 01001	rx	immediate
5	3	8

Format: `ADDIU rx, immediate` **MIPS16e**

Purpose:

To add a constant to a 32-bit integer.

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

The 8-bit *immediate* is sign-extended and then added to the contents of GPR *rx* to form a 32-bit result. The result is sign-extended and placed in GPR *rx*.

No integer overflow exception occurs under any circumstances.

Restrictions:

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

Operation:

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

Exceptions:

None

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Add Immediate Unsigned Word (2-Operand, Extended) ADDIU

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	imm 10:5	imm 15:11	ADDIU8 01001	rx	0 000	imm 4:0
5	6	5	5	3	3	5

Format: `ADDIU rx, immediate` **MIPS16e**

Purpose:

To add a constant to a 32-bit integer.

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

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

No integer overflow exception occurs under any circumstances.

Restrictions:

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

Operation:

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

Exceptions:

None

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Add Immediate Unsigned Word (3-Operand) ADDIU

15–11	10–8	7–5	4	3–0
RRI-A 01000	rx	ry	ADDIU 0	immediate
5	3	3	1	4

Format: `ADDIU ry, rx, immediate` **MIPS16e**

Purpose:

To add a constant to a 32-bit integer.

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

The 4-bit *immediate* is sign-extended and then added to the contents of GPR *rx* to form a 32-bit result. The result is sign-extended and placed into GPR *ry*.

No integer overflow exception occurs under any circumstances.

Restrictions:

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

Operation:

```
if (NotWordValue(GPR[Xlat(rx)])) then
    UNPREDICTABLE
endif
temp ← GPR[Xlat(rx)] + sign_extend(immediate)
GPR[Xlat(ry)] ← sign_extend(temp31..0)
```

Exceptions:

None

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Add Immediate Unsigned Word (3-Operand, Extended) ADDIU

31–27	26–20	19–16	15–11	10–8	7–5	4	3–0
EXTEND 11110	imm 10:4	imm 14:11	RRI-A 01000	rx	ry	ADDIU 0	imm 3:0
5	7	4	5	3	3	1	4

Format: `ADDIU ry, rx, immediate` **MIPS16e**

Purpose:

To add a constant to a 32-bit integer.

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

The 15-bit *immediate* is sign-extended and then added to the contents of GPR *rx* to form a 32-bit result. The result is sign-extended and placed into GPR *ry*.

No integer overflow exception occurs under any circumstances.

Restrictions:

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

Operation:

```
if (NotWordValue(GPR[Xlat(rx)])) then
    UNPREDICTABLE
endif
temp ← GPR[Xlat(rx)] + sign_extend(immediate)
GPR[Xlat(ry)] ← sign_extend(temp31..0)
```

Exceptions:

None

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Add Immediate Unsigned Word (3-Operand, PC-Relative) — ADDIU

15 … 11	10 … 8	7 … 0
ADDIUPC 00001	rx	immediate
5	3	8

Format: `ADDIU rx, pc, immediate` **MIPS16e**

Purpose:

To add a constant to the program counter.

Description: `GPR[rx] ← PC + (immediate << 2)`

The 8-bit *immediate* is shifted left two bits, zero-extended, and added to either the address of the ADDIU instruction or the address of the jump instruction in whose delay slot the ADDIU is executed. This result (with its two lower bits cleared) is sign-extended and placed in GPR *rx*.

No integer overflow exception occurs under any circumstances.

Restrictions:

If the base PC is outside the 32-bit Compatibility Address Space (i.e., bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```
I-1:  base_pc ← PC
I:    if not (JumpDelaySlot(PC)) then
         base_pc ← PC
      endif
      if NotWordValue(base_pc) then
         UNPREDICTABLE
      endif
      temp ← (base_pc_{GPRLEN-1..2} + zero_extend(immediate)) || 0^2)
      GPR[Xlat(rx)] ← sign_extend(temp_{31..0})
```

Exceptions:

None

Programming Notes:

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

The use of the ADDIUPC instruction on a MIPS64 processor in which the PC is outside the 32-bit Compatibility Address Space will not produce the expected result. This is because the final PC value is required to be sign-extended from the least-significant 32 bits, and such a value will not generate the correct address if PC is not also a sign-extended value. In such cases, DADDIUPC should be used instead.

Since the 8-bit *immediate* is shifted left two bits before being added to the PC, the range is 0, 4, 8..1020.

The assembler LA (Load Address) pseudo-instruction is implemented as a PC-relative add (using ADDIUPC for MIPS32 or DADDIUPC for MIPS64 code).

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Add Immediate Unsigned Word (3-Operand, PC-Relative, Extended) **ADDIU**

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	imm 10:5	imm 15:11	ADDIUPC 00001	rx	0 000	imm 4:0
5	6	5	5	3	3	5

Format: `ADDIU rx, pc, immediate` **MIPS16e**

Purpose:

To add a constant to the program counter.

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

The 16-bit *immediate* is sign-extended and added to the address of the ADDIU instruction. Before the addition, the two lower bits of the instruction address are cleared.

The result of the addition is sign-extended and placed in GPR *rx*.

No integer overflow exception occurs under any circumstances.

Restrictions:

A PC-relative, extended ADDIU may not be placed in the delay slot of a jump instruction.

If the PC is outside the 32-bit Compatibility Address Space (i.e., bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```
if NotWordValue(PC) then
    UNPREDICTABLE
endif
    temp ← (PC_GPRLEN-1..2 || 0^2) + sign_extend(immediate)
    GPR[Xlat(rx)] ← sign_extend(temp_31..0)
```

Exceptions:

None

Programming Notes:

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

The use of the ADDIUPC instruction on a MIPS64 processor in which the PC is outside the 32-bit Compatibility Address Space will not produce the expected result. This is because the final PC value is required to be sign-extended from the least-significant 32 bits, and such a value will not generate the correct address if PC is not also a sign-extended value. In such cases, DADDIUPC should be used instead.

The assembler LA (Load Address) pseudo-instruction is implemented as a PC-relative add (using ADDIUPC for MIPS32 or DADDIUPC for MIPS64 code).

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Add Immediate Unsigned Word (2-Operand, SP-Relative) — ADDIU

15 — 11	10 — 8	7 — 0
I8 01100	ADJSP 011	immediate
5	3	8

Format: `ADDIU sp, immediate` **MIPS16e**

Purpose:

To add a constant to the stack pointer.

Description: `GPR[sp] ← GPR[sp] + immediate`

The 8-bit *immediate* is shifted left three bits, sign-extended, and then added to the contents of GPR 29 to form a 32-bit result. The result is sign-extended and placed in GPR 29.

No integer overflow exception occurs under any circumstances.

Restrictions:

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

Operation:

```
if (NotWordValue(GPR[29])) then
    UNPREDICTABLE
endif
temp ← GPR[29] + sign_extend(immediate || 0^3)
GPR[29] ← sign_extend(temp_31..0)
```

Exceptions:

None

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Add Immediate Unsigned Word (2-Operand, SP-Relative, Extended) **ADDIU**

31 27	26 21	20 16	15 11	10 8	7 5	4 0
EXTEND 11110	imm 10:5	imm 15:11	I8 01100	ADJSP 011	0 000	imm 4:0
5	6	5	5	3	3	5

Format: `ADDIU sp, immediate` **MIPS16e**

Purpose:

To add a constant to the stack pointer.

Description: `GPR[sp] ← GPR[sp] + immediate`

The 16-bit *immediate* is sign-extended, and then added to the contents of GPR 29 to form a 32-bit result. The result is sign-extended and placed in GPR 29.

No integer overflow exception occurs under any circumstances.

Restrictions:

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

Operation:

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

Exceptions:

None

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Add Immediate Unsigned Word (3-Operand, SP-Relative) — ADDIU

15 – 11	10 – 8	7 – 0
ADDIUSP 00000	rx	immediate
5	3	8

Format: `ADDIU rx, sp, immediate` **MIPS16e**

Purpose:

To add a constant to the stack pointer.

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

The 8-bit *immediate* is shifted left two bits, zero-extended, and then added to the contents of GPR 29 to form a 32-bit result. The result is sign-extended and placed in GPR *rx*.

No integer overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
if (NotWordValue(GPR[29])) then
    UNPREDICTABLE
endif
temp ← GPR[29] + zero_extend(immediate || 0^2)
GPR[Xlat(rx)] ← sign_extend(temp_31..0)
```

Exceptions:

None

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Add Immediate Unsigned Word (3-Operand, SP-Relative, Extended) — ADDIU

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	imm 10:5	imm 15:11	ADDIUSP 00000	rx	0 000	imm 4:0
5	6	5	5	3	3	5

Format: `ADDIU rx, sp, immediate` **MIPS16e**

Purpose:

To add a constant to the stack pointer.

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

The 16-bit *immediate* is sign-extended and then added to the contents of GPR 29 to form a 32-bit result. The result is sign-extended and placed in GPR *rx*.

No integer overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
if (NotWordValue(GPR[29])) then
    UNPREDICTABLE
endif
temp ← GPR[29] + sign_extend(immediate
GPR[Xlat(rx)] ← sign_extend(temp31..0)
```

Exceptions:

None

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Add Unsigned Word (3-Operand) ADDU

15–11	10–8	7–5	4–2	1–0
RRR 11100	rx	ry	rz	ADDU 01
5	3	3	3	2

Format: `ADDU rz, rx, ry` **MIPS16e**

Purpose:

To add 32-bit integers.

Description: `GPR[rz] ← GPR[rx] + GPR[ry]`

The contents of GPR *rx* and GPR *ry* are added together to form a 32-bit result. The result is sign-extended and placed into GPR *rz*.

No integer overflow exception occurs under any circumstances.

Restrictions:

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

Operation:

```
if NotWordValue(GPR[Xlat(rx)]) or NotWordValue(GPR[Xlat(ry)]) then
    UNPREDICTABLE
endif
temp ← GPR[Xlat(rx)] + GPR[Xlat(ry)]
GPR[Xlat(rz)] ← sign_extend(temp31..0)
```

Exceptions:

None

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

15 … 11	10 … 8	7 … 5	4 … 0
RR 11101	rx	ry	AND 01100
5	3	3	5

Format: `AND rx, ry` **MIPS16e**

Purpose:

To do a bitwise logical AND.

Description: `GPR[rx] ← GPR[rx] AND GPR[ry]`

The contents of GPR *ry* are combined with the contents of GPR *rx* in a bitwise logical AND operation. The result is placed in GPR *rx*.

Restrictions:

None

Operation:

```
GPR[Xlat(rx)] ← GPR[Xlat(rx)] and GPR[Xlat(ry)]
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Application-Specific Macro Instructions — ASMACRO

31–27	26–24	23–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	select	p4	p3	RRR 11100	p2	p1	p0
5	3	3	5	5	3	3	5

Format: `ASMACRO select,p0,p1,p2,p3,p4` **MIPS16e**

The format listed is the most generic assembler format and is unlikely to be used for an actual implementation of application-specific macro instructions. Rather, the assembler format is likely to represent the use of the macro, with the assembler turning that format into the appropriate bit pattern required by the instruction.

Purpose:

To execute an implementation-definable macro instruction.

Description:

The ASMACRO instruction is the programming interface to the implementation-definable macro instruction facility that is defined by the MIPS16e architecture.

The *select* field specifies which of 8 possible macros is expanded. The definition of each macro specifies how the parameters *p0*, *p1*, *p2*, *p3*, and *p4* are substituted into the 32-bit instructions with which the macro is defined. The execution of the 32-bit instructions occurs while PC remains unchanged.

It is implementation-dependent whether a processor implements any implementation-definable macro instructions and, if it does, how many. It is implementation-dependent whether the macro is executed with interrupts disabled.

Restrictions:

The 32-bit instructions with which the macro is defined must by chosen with care. Issues of atomicity, restartability of the instruction sequence, and similar factors must be considered when using the implementation-definable macro instruction facility. Failure to do so can cause **UNPREDICTABLE** behavior.

If implementation-definable macro instructions are not implemented by the processor, or if the *select* field references a specific macro which is not implemented by the processor, a Reserved Instruction exception is signaled.

Operation:

```
ExecuteMacro(sel,p0,p1,p2,p3,p4)
```

Exceptions:

Reserved Instruction
Others as may be generated by the 32-bit instructions included in each macro expansion.

Programming Notes:

Implementations may impose certain restrictions on 32-bit instructions are supported within an ASMACRO instruction. For instance, many implementations may not allow loads, stores, branches or jumps within an ASMACRO definition. Refer to the Users Guide for each processor which implements this capability for a list of macros defined and implemented by that processor, and for any specific restrictions imposed by that processor.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Unconditional Branch **B**

15 — 11	10 — 0
B 00010	offset
5	11

Format: `B offset` **MIPS16e**

Purpose:

To do an unconditional PC-relative branch.

Description: `branch`

The 11-bit *offset* is shifted left 1 bit, sign-extended, and then added to the address of the instruction after the branch to form the target address. The program branches to the target address unconditionally.

Restrictions:

None

Operation:

```
I:    PC ← PC + 2 + sign_extend(offset || 0)
```

Exceptions:

None

Programming Notes:

In MIPS16e mode, the branch *offset* is interpreted as halfword-aligned. This is unlike 32-bit MIPS mode, which interprets the *offset* value as word-aligned.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Unconditional Branch (Extended) B

31–27	26–21	20–16	15–11	10–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	B 00010	0 000000	offset 4:0
5	6	5	5	6	5

Format: `B offset` **MIPS16e**

Purpose:

To do an unconditional PC-relative branch.

Description: `branch`

The 16-bit *offset* is shifted left 1 bit, sign-extended, and then added to the address of the instruction after the branch to form the target address. The program branches to the target address unconditionally.

Restrictions:

None

Operation:

```
I:     PC ← PC + 4 + sign_extend(offset || 0)
```

Exceptions:

None

Programming Notes:

In MIPS16e mode, the branch *offset* is interpreted as halfword-aligned. This is unlike 32-bit MIPS mode, which interprets the *offset* value as word-aligned.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Branch on Equal to Zero **BEQZ**

15 — 11	10 — 8	7 — 0
BEQZ 00100	rx	offset
5	3	8

Format: `BEQZ rx, offset` **MIPS16e**

Purpose:

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

Description: `if (GPR[rx] = 0) then branch`

The 8-bit *offset* is shifted left 1 bit, sign-extended, and then added to the address of the instruction after the branch to form the target address. If the contents of GPR *rx* are equal to zero, the program branches to the target address.

Restrictions:

None

Operation:

```
I:     tgt_offset ← sign_extend(offset || 0)
       condition ← (GPR[Xlat(rx)] = 0^GPRLEN)
       if condition then
          PC ← PC + 2 + tgt_offset
       endif
```

Exceptions:

None

Programming Notes:

In MIPS16e mode, the branch *offset* is interpreted as halfword-aligned. This is unlike 32-bit MIPS mode, which interprets the *offset* value as word-aligned.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Branch on Equal to Zero (Extended) BEQZ

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	BEQZ 00100	rx	0 000	offset 4:0
5	6	5	5	3	3	5

Format: `BEQZ rx, offset` **MIPS16e**

Purpose:

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

Description: `if (GPR[rx] = 0) then branch`

The 16-bit *offset* is shifted left 1 bit, sign-extended, and then added to the address of the instruction after the branch to form the target address. If the contents of GPR *rx* are equal to zero, the program branches to the target address.

Restrictions:

None

Operation:

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

Exceptions:

None

Programming Notes:

In MIPS16e mode, the branch *offset* is interpreted as halfword-aligned. This is unlike 32-bit MIPS mode, which interprets the *offset* value as word-aligned.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Branch on Not Equal to Zero **BNEZ**

15 ... 11	10 ... 8	7 ... 0
BNEZ 00101	rx	offset
5	3	8

Format: `BNEZ rx, offset` **MIPS16e**

Purpose:

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

Description: `if (GPR[rx] ≠ 0) then branch`

The 8-bit *offset* is shifted left 1 bit, sign-extended, and then added to the address of the instruction after the branch to form the target address. If the contents of GPR *rx* are not equal to zero, the program branches to the target address.

Restrictions:

None

Operation:

```
I:     tgt_offset ← sign_extend(offset || 0)
       condition ← (GPR[Xlat(rx)] ≠ 0^GPRLEN)
       if condition then
          PC ← PC + 2 + tgt_offset
       endif
```

Exceptions:

None

Programming Notes:

In MIPS16e mode, the branch *offset* is interpreted as halfword-aligned. This is unlike 32-bit MIPS mode, which interprets the *offset* value as word-aligned.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Branch on Not Equal to Zero (Extended) — BNEZ

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	BNEZ 00101	rx	0 000	offset 4:0
5	6	5	5	3	3	5

Format: `BNEZ rx, offset` **MIPS16e**

Purpose:

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

Description: `if (GPR[rx] ≠ 0) then branch`

The 16-bit *offset* is shifted left 1 bit, sign-extended, and then added to the address of the instruction after the branch to form the target address. If the contents of GPR *rx* are not equal to zero, the program branches to the target address.

Restrictions:

None

Operation:

```
I:     tgt_offset ← sign_extend(offset || 0)
       condition ← (GPR[Xlat(rx)] ≠ 0^GPRLEN)
       if condition then
          PC ← PC + 4 + tgt_offset
       endif
```

Exceptions:

None

Programming Notes:

In MIPS16e mode, the branch *offset* is interpreted as halfword-aligned. This is unlike 32-bit MIPS mode, which interprets the *offset* value as word-aligned.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Breakpoint BREAK

15 … 11	10 … 8 7 … 5	4 … 0
RR 11101	code	BREAK 00101
5	6	5

Format: `BREAK immediate` **MIPS16e**

Purpose:

To cause a Breakpoint exception.

Description:

A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.

Restrictions:

None

Operation:

```
SignalException(Breakpoint)
```

Exceptions:

Breakpoint

Programming Notes:

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory halfword containing the instruction.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Branch on T Equal to Zero **BTEQZ**

15 … 11	10 … 8	7 … 0
I8 01100	BTEQZ 000	offset
5	3	8

Format: `BTEQZ offset` **MIPS16e**

Purpose:

To test special register T then do a PC-relative conditional branch.

Description: `if (T = 0) then branch`

The 8-bit *offset* is shifted left 1 bit, sign-extended, and then added to the address of the instruction after the branch to form the target address. If the contents of GPR 24 are equal to zero, the program branches to the target address.

Restrictions:

None

Operation:

```
I:     tgt_offset ← sign_extend(offset || 0)
       condition ← (GPR[24] = 0^GPRLEN)
        if condition then
          PC ← PC + 2 + tgt_offset
       endif
```

Exceptions:

None

Programming Notes:

In MIPS16e mode, the branch *offset* is interpreted as halfword-aligned. This is unlike 32-bit MIPS mode, which interprets the *offset* value as word-aligned.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Branch on T Equal to Zero (Extended) BTEQZ

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	I8 01100	BTEQZ 000	000 0	offset 4:0
5	6	5	5	3	3	5

Format: `BTEQZ offset` **MIPS16e**

Purpose:

To test special register T then do a PC-relative conditional branch.

Description: `if (T = 0) then branch`

The 16-bit *offset* is shifted left 1 bit, sign-extended, and then added to the address of the instruction after the branch to form the target address. If the contents of GPR 24 are equal to zero, the program branches to the target address.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend(offset || 0)
      condition ← (GPR[24] = 0^GPRLEN)
      if condition then
         PC ← PC + 4 + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

In MIPS16e mode, the branch *offset* is interpreted as halfword-aligned. This is unlike 32-bit MIPS mode, which interprets the *offset* value as word-aligned.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

15–11	10–8	7–0
I8 01100	BTNEZ 001	offset
5	3	8

Format: `BTNEZ offset` **MIPS16e**

Purpose:

To test special register T then do a PC-relative conditional branch.

Description: `if (T ≠ 0) then branch`

The 8-bit *offset* is shifted left 1 bit, sign-extended, and then added to the address of the instruction after the branch to form the target address. If the contents of GPR 24 are not equal to zero, the program branches to the target address.

Restrictions:

None

Operation:

```
I:     tgt_offset ← sign_extend(offset || 0)
       condition ← (GPR[24] ≠ 0^GPRLEN)
       if condition then
          PC ← PC + 2 + tgt_offset
       endif
```

Exceptions:

None

Programming Notes:

In MIPS16e mode, the branch *offset* is interpreted as halfword-aligned. This is unlike 32-bit MIPS mode, which interprets the *offset* value as word-aligned.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Branch on T Not Equal to Zero (Extended) BTNEZ

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	I8 01100	BTNEZ 001	000 0	offset 4:0
5	6	5	5	3	3	5

Format: `BTNEZ offset` **MIPS16e**

Purpose:

To test special register T then do a PC-relative conditional branch.

Description: `if (T ≠ 0) then branch`

The 16-bit *offset* is shifted left 1 bit, sign-extended, and then added to the address of the instruction after the branch to form the target address. If the contents of GPR 24 are not equal to zero, the program branches to the target address.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend(offset || 0)
      condition ← (GPR[24] ≠ 0^GPRLEN)
      if condition then
         PC ← PC + 4 + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

In MIPS16e mode, the branch *offset* is interpreted as halfword-aligned. This is unlike 32-bit MIPS mode, which interprets the *offset* value as word-aligned.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Compare **CMP**

15 — 11	10 — 8	7 — 5	4 — 0
RR 11101	rx	ry	CMP 01010
5	3	3	5

Format: `CMP rx, ry` **MIPS16e**

Purpose:

To compare the contents of two GPRs.

Description: `T ← GPR[rx] XOR GPR[ry]`

The contents of GPR *ry* are Exclusive-ORed with the contents of GPR *rx*. The result is placed into GPR 24.

Restrictions:

None

Operation:

```
GPR[24] ← GPR[Xlat(ry)] xor GPR[Xlat(rx)]
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Compare Immediate **CMPI**

15 — 11	10 — 8	7 — 0
CMPI 01110	rx	immediate
5	3	8

Format: `CMPI rx, immediate` **MIPS16e**

Purpose:

To compare a constant with the contents of a GPR.

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

The 8-bit *immediate* is zero-extended and Exclusive-ORed with the contents of GPR *rx*. The result is placed into GPR 24.

Restrictions:

None

Operation:

```
GPR[24] ← GPR[Xlat(rx)] xor zero_extend(immediate)
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Compare Immediate (Extended) CMPI

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	imm 10:5	imm 15:11	CMPI 01110	rx	000 0	imm 4:0
5	6	5	5	3	3	5

Format: `CMPI rx, immediate` **MIPS16e**

Purpose:

To compare a constant with the contents of a GPR.

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

The 16-bit *immediate* is zero-extended and Exclusive-ORed with the contents of GPR *rx*. The result is placed into GPR 24.

Restrictions:

None

Operation:

```
GPR[24] ← GPR[Xlat(rx)] xor zero_extend(immediate)
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Add Immediate Unsigned (2-Operand) DADDIU

15–11	10–8	7–5	4–0
I64 11111	DADDIU5 101	ry	immediate
5	3	3	5

Format: `DADDIU ry, immediate` **MIPS16e (64-bit only)**

Purpose:

To add a constant to a 64-bit integer.

Description: `GPR[ry] ← GPR[ry] + immediate`

The 5-bit *immediate* is sign-extended to 64 bits and then added to the contents of GPR *ry* to form a 64-bit result. The result is placed in GPR *ry*.

No integer overflow exception occurs under any circumstances.

Restrictions:

Operation:

```
GPR[Xlat(ry)] ← GPR[Xlat(ry)] + sign_extend(immediate)
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Add Immediate Unsigned (2-Operand, Extended) DADDIU

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	imm 10:5	imm 15:11	I64 11111	DADDIU5 101	ry	imm 4:0
5	6	5	5	3	3	5

Format: `DADDIU ry, immediate` **MIPS16e (64-bit only)**

Purpose:

To add a constant to a 64-bit integer.

Description: `GPR[ry] ← GPR[ry] + immediate`

The 16-bit *immediate* is sign-extended to 64 bits and then added to the contents of GPR *ry* to form a 64-bit result. The result is placed in GPR *ry*.

No integer overflow exception occurs under any circumstances.

Restrictions:

Operation:

```
GPR[Xlat(ry)] ← GPR[Xlat(ry)] + sign_extend(immediate)
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Add Immediate Unsigned (3-Operand) — DADDIU

15 — 11	10 — 8	7 — 5	4	3 — 0
RRI-A 01000	rx	ry	DADDIU 1	immediate
5	3	3	1	4

Format: `DADDIU ry, rx, immediate` **MIPS16e (64-bit only)**

Purpose:

To add a constant to a 64-bit integer.

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

The 4-bit *immediate* is sign-extended to 64 bits and then added to the contents of GPR *rx* to form a 64-bit result. The result is placed in GPR *ry*.

No integer overflow exception occurs under any circumstances.

Restrictions:

Operation:

```
GPR[Xlat(ry)] ← GPR[Xlat(rx)] + sign_extend(immediate)
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Add Immediate Unsigned (3-Operand, Extended) **DADDIU**

31–27	26–20	19–16	15–11	10–8	7–5	4	3–0
EXTEND 11110	imm 10:4	imm 14:11	RRI-A 01000	rx	ry	DADDIU 1	imm 3:0
5	7	4	5	3	3	1	4

Format: `DADDIU ry, rx, immediate` **MIPS16e (64-bit only)**

Purpose:

To add a constant to a 64-bit integer.

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

The 15-bit *immediate* is sign-extended to 64 bits and then added to the contents of GPR *rx* to form a 64-bit result. The result is placed in GPR *ry*.

No integer overflow exception occurs under any circumstances.

Restrictions:

Operation:

```
GPR[Xlat(ry)] ← GPR[Xlat(rx)] + sign_extend(immediate)
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Add Immediate Unsigned (3-Operand, PC-Relative) DADDIU

15 — 11	10 — 8	7 — 5	4 — 0
I64 11111	DADDIUPC 110	ry	immediate
5	3	3	5

Format: `DADDIU ry, pc, immediate` **MIPS16e (64-bit only)**

Purpose:

To add a constant to the program counter.

Description: `GPR[ry] ← PC + (immediate << 2)`

The 5-bit *immediate* is shifted left 2 bits, zero-extended, and added either to the address of the DADDIU instruction or to the address of the jump instruction in whose delay slot the DADDIU is executed. This result (with its 2 lower bits cleared) is placed in GPR *ry*.

No integer overflow exception occurs under any circumstances.

Restrictions:

Operation:

```
I-1:  base_pc ← PC
I:    if not (JumpDelaySlot(PC)) then
         base_pc ← PC
      endif
      GPR[Xlat(ry)] ← (base_pc_{GPRLEN-1..2} + zero_extend(immediate)) || 0^2
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Add Immediate Unsigned (3-Operand, PC-Relative, Extended) — DADDIU

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	imm 10:5	imm 15:11	I64 11111	DADDIUPC 110	ry	imm 4:0
5	6	5	5	3	3	5

Format: `DADDIU ry, pc, immediate` **MIPS16e (64-bit only)**

Purpose:

To add a constant to the program counter.

Description: `GPR[ry] ← PC + immediate`

The 16-bit *immediate* is sign-extended and added to the address of the DADDIU instruction. Before the addition, the two lower bits of the instruction address are cleared. The result of the addition is placed in GPR *ry*.

No integer overflow exception occurs under any circumstances.

Restrictions:

A PC-relative extended DADDIU may not be placed in the delay slot of a jump instruction.

Operation:

```
temp ← (PC_{GPRLEN-1..2} || 0^2) + sign_extend(immediate)
GPR[Xlat(ry)] ← temp_{63..0}
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Add Immediate Unsigned (2-Operand, SP-Relative) **DADDIU**

15 — 11	10 — 8	7 — 0
I64 11111	DADJSP 011	immediate
5	3	8

Format: `DADDIU sp, immediate` **MIPS16e (64-bit only)**

Purpose:

To add a constant to the stack pointer.

Description: `GPR[sp] ← GPR[sp] + immediate`

The 8-bit *immediate* is shifted left 3 bits, sign-extended to 64 bits, and then added to the contents of GPR 29 to form a 64-bit result. The result is placed in GPR 29.

No integer overflow exception occurs under any circumstances.

Restrictions:

Operation:

$$\texttt{GPR[29]} \leftarrow \texttt{GPR[29] + sign\_extend(immediate || } 0^3\texttt{)}$$

Exceptions:

Reserved Instruction

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Add Immediate Unsigned (2-Operand, SP-Relative, Extended) **DADDIU**

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	imm 10:5	imm 15:11	I64 11111	DADDJSP 011	000 0	imm 4:0
5	6	5	5	3	3	5

Format: `DADDIU sp, immediate` **MIPS16e (64-bit only)**

Purpose:

To add a constant to the stack pointer.

Description: `GPR[sp] ← GPR[sp] + immediate`

The 16-bit *immediate* is sign-extended to 64 bits and then added to the contents of GPR 29 to form a 64-bit result. The result is placed in GPR 29.

No integer overflow exception occurs under any circumstances.

Restrictions:

Operation:

```
GPR[29] ← GPR[29] + sign_extend(immediate)
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Add Immediate Unsigned (3-Operand, SP-Relative) **DADDIU**

15–11	10–8	7–5	4–0
I64 11111	DADDIUSP 111	ry	immediate
5	3	3	5

Format: `DADDIU ry, sp, immediate` **MIPS16e (64-bit only)**

Purpose:

To add a constant to the stack pointer.

Description: `GPR[ry] ← GPR[sp] + immediate`

The 5-bit *immediate* is shifted left 2 bits, zero-extended to 64 bits, and added to the contents of GPR 29 to form a 64-bit result. The result is placed in GPR *ry*.

No integer overflow exception occurs under any circumstances.

Restrictions:

Operation:

$$GPR[Xlat(ry)] \leftarrow GPR[29] + zero\_extend(immediate \mid\mid 0^{2})$$

Exceptions:

Reserved Instruction

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Add Immediate Unsigned (3-Operand, SP-Relative, Extended) DADDIU

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	imm 10:5	imm 15:11	I64 11111	DADDIUSP 111	ry	imm 4:0
5	6	5	5	3	3	5

Format: `DADDIU ry, sp, immediate` **MIPS16e (64-bit only)**

Purpose:

To add a constant to the stack pointer.

Description: `GPR[ry] ← GPR[sp] + immediate`

The 16-bit *immediate* is sign-extended to 64 bits and added to the contents of GPR 29 to form a 64-bit result. The result is placed in GPR *ry*.

No integer overflow exception occurs under any circumstances.

Restrictions:

Operation:

```
GPR[Xlat(ry)] ← GPR[29] + sign_extend(immediate)
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Add Unsigned (3-Operand) **DADDU**

15–11	10–8	7–5	4–2	1–0
RRR 11100	rx	ry	rz	DADDU 00
5	3	3	3	2

Format: `DADDU rz, rx, ry` **MIPS16e (64-bit only)**

Purpose:

To add 64-bit integers.

Description: `GPR[rz] ← GPR[rx] + GPR[ry]`

The contents of GPR *ry* are added to the contents of GPR *rx*. The 64-bit result is placed into GPR *rz*.

No integer overflow exception occurs under any circumstances.

Restrictions:

Operation:

```
GPR[Xlat(rz)] ← GPR[Xlat(rx)] + GPR[Xlat(ry)]
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Divide DDIV

15 ... 11	10 ... 8	7 ... 5	4 ... 0
RR 11101	rx	ry	DDIV 11110
5	3	3	5

Format: `DDIV rx, ry` **MIPS16e (64-bit only)**

Purpose:

To divide 64-bit signed integers.

Description: `(LO, HI) ← GPR[rx] / GPR[ry]`

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

No arithmetic exception occurs under any circumstances.

Restrictions:

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

Operation:

```
LO ← GPR[Xlat(rx)] div GPR[Xlat(ry)]
HI ← GPR[Xlat(rx)] mod GPR[Xlat(ry)]
```

Exceptions:

Reserved Instruction

Programming Notes:

See “Programming Notes” for the DIV instruction.

Historical Perspective:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Divide Unsigned — DDIVU

15–11	10–8	7–5	4–0
RR 11101	rx	ry	DDIVU 11111
5	3	3	5

Format: `DDIVU rx, ry` **MIPS16e (64-bit only)**

Purpose:

To divide 64-bit unsigned integers.

Description: `(LO, HI) ← GPR[rx] / GPR[ry]`

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

No arithmetic exception occurs under any circumstances.

Restrictions:

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

Operation:

```
q ← (0 || GPR[Xlat(rx)]) div (0 || GPR[Xlat(ry)])
r ← (0 || GPR[Xlat(rx)]) mod (0 || GPR[Xlat(ry)])
LO ← q63..0
HI ← r63..0
```

Exceptions:

Reserved Instruction

Programming Notes:

See “Programming Notes” for the DIV instruction.

Historical Perspective:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Divide Word **DIV**

15–11	10–8	7–5	4–0
RR 11101	rx	ry	DIV 11010
5	3	3	5

Format: `DIV rx, ry` **MIPS16e**

Purpose:

To divide 32-bit signed integers.

Description: `(LO, HI) ← GPR[rx] / GPR[ry]`

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

No arithmetic exception occurs under any circumstances.

Restrictions:

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

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

Operation:

```
if (NotWordValue(GPR[Xlat(rx)]) or NotWordValue(GPR[Xlat(ry)])) then
    UNPREDICTABLE
endif
q  ← GPR[Xlat(rx)] div GPR[Xlat(ry)]
r  ← GPR[Xlat(rx)] mod GPR[Xlat(ry)]
LO ← sign_extend(q31..0)
HI ← sign_extend(r31..0)
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Programming Notes:

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

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

Where the size of the operands are known, software should place the shorter operand in GPR *ry*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

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

Historical Perspective:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Divide Unsigned Word **DIVU**

15 — 11	10 — 8	7 — 5	4 — 0
RR 11101	rx	ry	DIVU 11011
5	3	3	5

Format: `DIVU rx, ry` **MIPS16e**

Purpose:

To divide 32-bit unsigned integers.

Description: `(LO, HI) ← GPR[rx] / GPR[ry]`

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

Restrictions:

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

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

Operation:

```
if (NotWordValue(GPR[Xlat(rx)]) or NotWordValue(GPR[Xlat(ry)])) then
    UNPREDICTABLE
endif
q ← (0 || GPR[Xlat(rx)]) div (0 || GPR[Xlat(ry)])
r ← (0 || GPR[Xlat(rx)]) mod (0 || GPR[Xlat(ry)])
LO ← sign_extend(q31..0)
HI ← sign_extend(r31..0)
```

Exceptions:

None

Programming Notes:

See “Programming Notes” for the DIV instruction.

Historical Perspective:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Multiply — DMULT

15–11	10–8	7–5	4–0
RR 11101	rx	ry	DMULT 11100
5	3	3	5

Format: `DMULT rx, ry` **MIPS16e (64-bit only)**

Purpose:

To multiply 64-bit signed integers.

Description: `(LO, HI) ← GPR[rx] × GPR[ry]`

The 64-bit doubleword value in GPR *rx* is multiplied by the 64-bit value in GPR *ry*, treating both operands as signed values, to produce a 128-bit result. The low-order 64-bit doubleword of the result is placed into special register *LO*, and the high-order 64-bit doubleword is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

Operation:

```
prod← GPR[Xlat(rx)] × GPR[Xlat(ry)]
LO  ← prod63..0
HI  ← prod127..64
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Programs that require overflow detection must check for it explicitly.

Historical Perspective:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Multiply Unsigned — DMULTU

15–11	10–8	7–5	4–0
RR 11101	rx	ry	DMULTU 11101
5	3	3	5

Format: `DMULTU rx, ry` **MIPS16e (64-bit only)**

Purpose:

To multiply 64-bit unsigned integers.

Description: `(LO, HI)` $\leftarrow$ `GPR[rx]` $\times$ `GPR[ry]`

The 64-bit doubleword value in GPR *rx* is multiplied by the 64-bit value in GPR *ry*, treating both operands as unsigned values, to produce a 128-bit result. The low-order 64-bit doubleword of the result is placed into special register *LO*, and the high-order 64-bit doubleword is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

Operation:

```
prod← (0||GPR[Xlat(rx)]) × (0||GPR[Xlat(ry)])
LO  ← prod63..0
HI  ← prod127..64
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Programs that require overflow detection must check for it explicitly.

Historical Perspective:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

15–11	10–8	7–5	4–2	1–0
SHIFT 00110	rx	ry	sa	DSLL 01
5	3	3	3	2

Format: `DSLL rx, ry, sa` **MIPS16e (64-bit only)**

Purpose:

To execute a left-shift of a doubleword by a fixed amount—1 to 8 bits.

Description: `GPR[rx] ← GPR[ry] << sa`

The 64-bit doubleword contents of GPR *ry* are shifted left, and zeros are inserted into the emptied low-order bits. The 3-bit *sa* field specifies the shift amount. A shift amount of 0 is interpreted as a shift amount of 8. The 64-bit result is placed into GPR *rx*.

Restrictions:

Operation: 64-bit processors

```
if sa = 0^3 then
    s ← 8
else
    s ← 0^3 || sa
endif
GPR[Xlat(rx)] ← GPR[Xlat(ry)]_(63-s)..0 || 0^s
```

Exceptions:

Reserved Instruction

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Shift Left Logical (Extended) — DSLL

31–27	26–22	21	20–16	15–11	10–8	7–5	4–2	1–0
EXTEND 11110	sa 4:0	s5	0 00000	SHIFT 00110	rx	ry	0 000	DSLL 01
5	5	1	5	5	3	3	3	2

Format: `DSLL rx, ry, sa` **MIPS16e (64-bit only)**

Purpose:

To execute a left-shift of a doubleword by a fixed amount—0 to 63 bits.

Description: `GPR[rx] ← GPR[ry] << sa`

The 64-bit doubleword contents of GPR *ry* are shifted left, and zeros are inserted into the emptied low-order bits. The s_5 bit and the 5-bit *sa* field specify the effective 6-bit-shift amount. The 64-bit result is placed into GPR *rx*.

Restrictions:

None

Operation: 64-bit processors

```
s ← s5 || sa
GPR[Xlat(rx)] ← GPR[Xlat(ry)](63-s)..0 || 0^s
```

Exceptions:

Reserved Instruction

Programming Notes:

For DSLL only, the s_5 bit is the most-significant bit of the 6-bit-shift amount (*sa*) field. For all 32-bit extended shifts, s_5 must be zero. None of the extended shift instructions perform the zero-to-eight mapping, so zero-bit shifts are possible using the extended format.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Shift Left Logical Variable **DSLLV**

15 — 11	10 — 8	7 — 5	4 — 0
RR 11101	rx	ry	DSLLV 10100
5	3	3	5

Format: `DSLLV ry, rx` **MIPS16e (64-bit only)**

Purpose:

To execute a left-shift of a doubleword by a variable number of bits.

Description: `GPR[ry] ← GPR[ry] << GPR[rx]`

The 64-bit doubleword contents of GPR *ry* are shifted left, inserting zeros into the emptied bits; the result is placed back into GPR *ry*. The 6 low-order bits of GPR *rx* specify the shift amount.

Restrictions:

Operation: 64-bit processors

$s \leftarrow \mathrm{GPR[Xlat(rx)]}_{5..0}$
$\mathrm{GPR[Xlat(ry)]} \leftarrow \mathrm{GPR[Xlat(ry)]}_{(63-s)..0} \,||\, 0^{s}$

Exceptions:

Reserved Instruction

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Shift Right Arithmetic **DSRA**

15–11	10–8	7–5	4–0
RR 11101	sa	ry	DSRA 10011
5	3	3	5

Format: `DSRA ry, sa` **MIPS16e (64-bit only)**

Purpose:

To execute an arithmetic right-shift of a doubleword by a fixed amount—1 to 8 bits.

Description: `GPR[ry] ← GPR[ry] >> sa (arithmetic)`

The 64-bit doubleword contents of GPR *ry* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in back in GPR *ry*. The 3-bit *sa* field specifies the shift amount. A shift amount of 0 is interpreted as a shift amount of 8.

Restrictions:

Operation:

```
if sa = 0^3 then
    s ← 8
else
    s ← 0^3 || sa
endif
GPR[Xlat(ry)] ← (GPR[Xlat(ry)]_63)^s || GPR[Xlat(ry)]_63..s
```

Exceptions:

Reserved Instruction

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Shift Right Arithmetic (Extended) — DSRA

31–27	26–22	21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	sa 4:0	s5	0 00000	RR 11101	0 000	ry	DSRA 10011
5	5	1	5	5	3	3	5

Format: `DSRA ry, sa` **MIPS16e (64-bit only)**

Purpose:

To execute an arithmetic right-shift of a doubleword by a fixed amount—0 to 63 bits.

Description: `GPR[ry] ← GPR[ry] >> sa (arithmetic)`

The 64-bit doubleword contents of GPR *ry* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in back in GPR *ry*. The s_5 bit and the 5-bit *sa* field specify the effective 6-bit-shift amount.

Restrictions:

Operation:

$$s \leftarrow s5 \;||\; sa$$
$$GPR[Xlat(ry)] \leftarrow (GPR[Xlat(ry)]_{63})^{s} \;||\; GPR[Xlat(ry)]_{63..s}$$

Exceptions:

Reserved Instruction

Programming Notes:

The s_5 bit is the most-significant bit of the 6-bit-shift amount (*sa*) field. None of the extended shift instructions perform the zero-to-eight mapping, so zero-bit shifts are possible using the extended format.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Shift Right Arithmetic Variable **DSRAV**

15 … 11	10 … 8	7 … 5	4 … 0
RR 11101	rx	ry	DSRAV 10111
5	3	3	5

Format: `DSRAV ry, rx` **MIPS16e (64-bit only)**

Purpose:

To execute an arithmetic right-shift of a doubleword by a variable number of bits.

Description: `GPR[ry] ← GPR[ry] >> GPR[rx] (arithmetic)`

The doubleword contents of GPR *ry* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed back in GPR *ry*. The 6 low-order bits of GPR *rx* specify the shift amount.

Restrictions:

Operation:

$s \leftarrow \mathrm{GPR[Xlat(rx)]}_{5..0}$
$\mathrm{GPR[Xlat(ry)]} \leftarrow (\mathrm{GPR[Xlat(ry)]}_{63})^{s} \;||\; \mathrm{GPR[Xlat(ry)]}_{63..s}$

Exceptions:

Reserved Instruction

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Shift Right Logical — DSRL

15–11	10–8	7–5	4–0
RR 11101	sa	ry	DSRL 01000
5	3	3	5

Format: `DSRL ry, sa` **MIPS16e (64-bit only)**

Purpose:

To execute a logical right-shift of a doubleword by a fixed amount—1 to 8 bits.

Description: `GPR[ry] ← GPR[ry] >> sa (logical)`

The doubleword contents of GPR *ry* are shifted right, inserting zeros into the emptied bits; the result is placed back in GPR *ry*.The 3-bit *sa* field specifies the shift amount. A shift amount of 0 is interpreted as a shift amount of 8.

Restrictions:

Operation:

```
if sa = 0^3 then
    s ← 8
else
    s ← 0^3 || sa
endif
GPR[Xlat(ry)] ← 0^s || GPR[Xlat(ry)]_63..s
```

Exceptions:

Reserved Instruction

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Shift Right Logical (Extended) **DSRL**

31–27	26–22	21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	sa4:0	s5	0 00000	RR 11101	0 000	ry	DSRL 01000
5	5	1	5	5	3	3	5

Format: `DSRL ry, sa` **MIPS16e (64-bit only)**

Purpose:

To execute a logical right-shift of a doubleword by a fixed amount—0 to 63 bits

Description: `GPR[ry] ← GPR[ry] >> sa (logical)`

The doubleword contents of GPR *ry* are shifted right, inserting zeros into the emptied bits; the result is placed back in GPR *ry*. The s_5 bit and the 5-bit *sa* field specify the effective 6-bit-shift amount.

Restrictions:

Operation: 64-bit processors

```
s ← s5 || sa
GPR[Xlat(ry)] ← 0^s || GPR[Xlat(ry)]63..s
```

Exceptions:

Reserved Instruction

Programming Notes:

The s_5 bit is the most-significant bit of the 6-bit-shift amount (*sa*) field. None of the extended shift instructions perform the zero-to-eight mapping, so zero-bit shifts are possible using the extended format.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Shift Right Logical Variable **DSRLV**

15 ... 11	10 ... 8	7 ... 5	4 ... 0
RR 11101	rx	ry	DSRLV 10110
5	3	3	5

Format: `DSRLV ry, rx` **MIPS16e (64-bit only)**

Purpose:

To execute a logical right-shift of a doubleword by a variable number of bits.

Description: `GPR[ry] ← GPR[ry] >> GPR[rx] (logical)`

The 64-bit doubleword contents of GPR *ry* are shifted right, inserting zeros into the emptied bits; the result is placed back in GPR *ry*. The 6 low-order bits of GPR *rx* specify the shift amount.

Restrictions:

Operation: 64-bit processors

$$s \leftarrow GPR[Xlat(rx)]_{5..0}$$
$$GPR[Xlat(ry)] \leftarrow 0^{s} \,||\, GPR[Xlat(ry)]_{63..s}$$

Exceptions:

Reserved Instruction

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Doubleword Subtract Unsigned — DSUBU

15–11	10–8	7–5	4–2	1–0
RRR 11100	rx	ry	rz	DSUBU 10
5	3	3	3	2

Format: `DSUBU rz, rx, ry` **MIPS16e (64-bit only)**

Purpose:

To subtract 64-bit integers.

Description: `GPR[rz] ← GPR[rx] - GPR[ry]`

The 64-bit doubleword value in GPR *ry* is subtracted from the 64-bit value in GPR *rx* and the 64-bit arithmetic result is placed into GPR *rz*

No integer overflow exception occurs under any circumstances.

Restrictions:

Operation: 64-bit processors

```
GPR[Xlat(rz)] ← GPR[Xlat(rx)] - GPR[Xlat(ry)]
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

31 27	26	25 21	20 16	15 0
JAL 00011	x 0	target 20:16	target 25:21	target 15:0
5	1	5	5	16

Format: `JAL target` **MIPS16e**

Purpose:

To execute a procedure call within the current 256 MB-aligned region and preserve the current ISA.

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call. The value stored in GPR 31 bit 0 reflects the current value of the *ISA Mode* bit.

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

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

The opcode field describes a general jump-and-link operation, with the *x* field as a variable. The individual instructions, JAL and JALX have specific values for this variables.

Restrictions:

An extended instruction should not be placed in a jump delay slot as it causes one-half of an instruction to be executed.

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

Operation:

I: $GPR[31] \leftarrow (PC + 6)_{GPRLEN-1..1} \,||\, ISAMode$
I+1: $PC \leftarrow PC_{GPRLEN-1..28} \,||\, target \,||\, 0^2$

Exceptions:

None

Programming Notes:

Forming the jump target address by catenating PC and the 26-bit target address rather than adding a signed *offset* to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative *offset*.

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Jump and Link Register **JALR**

15–11	10–8	7	6	5	4–0
RR 11101	rx	nd 0	l 1	ra 0	J(AL)R(C) 00000
5	3	1	1	1	5

Format: `JALR ra, rx` **MIPS16e**

Purpose:

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

Description: `GPR[ra] ← return_addr, PC ← GPR[rx]`

The program unconditionally jumps to the address contained in GPR *rx*, with a delay of one instruction. The instruction sets the *ISA Mode* bit to the value in GPR *rx* bit 0.

The address of the instruction following the delay slot is placed into GPR 31. The value stored in GPR 31 bit 0 reflects the current value of the *ISA Mode* bit.

Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

The opcode and function field describe a general jump-thru-register operation, with the *nd* (no delay slot), *l* (link), and *ra* (source register is ra) fields as variables. The individual instructions, JALR, JR, JALRC, and JRC have specific values for these variables.

Restrictions:

The effective target address in GPR *rx* must be naturally-aligned. If bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

An extended instruction should not be placed in a jump delay slot, because this causes one-half of an instruction to be executed.

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

Operation:

```
I:      GPR[31] ← (PC + 4)GPRLEN-1..1 || ISAMode
I+1:    PC ← GPR[Xlat(rx)]GPRLEN-1..1 || 0
        ISAMode ← GPR[Xlat(rx)]0
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

15–11	10–8	7	6	5	4–0
RR 11101	rx	nd 1	l 1	ra 0	J(AL)R(C) 00000
5	3	1	1	1	5

Format: `JALRC ra, rx` **MIPS16e**

Purpose:

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

Description: `GPR[ra] ← return_addr, PC ← GPR[rx]`

The program unconditionally jumps to the address contained in GPR *rx*, with no delay slot instruction. The instruction sets the *ISA Mode* bit to the value in GPR *rx* bit 0.

The address of the instruction following the jump is placed into GPR 31. The value stored in GPR 31 bit 0 reflects the current value of the *ISA Mode* bit.

Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

The opcode and function field describe a general jump-thru-register operation, with the *nd* (no delay slot), *l* (link), and *ra* (source register is ra) fields as variables. The individual instructions, JALR, JR, JALRC, and JRC have specific values for these variables.

Restrictions:

The effective target address in GPR *rx* must be naturally-aligned. If bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

Operation:

```
I:     GPR[31] ← (PC + 2)GPRLEN-1..1 || ISAMode
       PC ← GPR[Xlat(rx)]GPRLEN-1..1 || 0
       ISAMode ← GPR[Xlat(rx)]0
```

Exceptions:

None.

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Jump and Link Exchange (MIPS16e Format) JALX

31–27	26	25–21	20–16	15–0
JAL 00011	x 1	target 20:16	target 25:21	target 15:0
5	1	5	5	16

Format: `JALX target` **MIPS16e**

Purpose:

To execute a procedure call within the current 256 MB-aligned region and change the ISA Mode from MIPS16e to 32-bit MIPS.

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call. The value stored in GPR 31 bit 0 reflects the current value of the *ISA Mode* bit.

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

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

The opcode field describes a general jump-and-link operation, with the *x* field as a variable. The individual instructions, JAL and JALX have specific values for this variables.

Restrictions:

An extended instruction should not be placed in a jump delay slot, because this causes one-half an instruction to be executed.

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

Operation:

```
I:     GPR[31] ← (PC + 6)GPRLEN-1..1 || ISAMode
I+1:   PC ← PCGPRLEN-1..28 || target || 0^2
       ISAMode ← (not ISAMode)
```

Exceptions:

None

Programming Notes:

Forming the jump target address by catenating PC and the 26-bit target address rather than adding a signed *offset* to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a jump to anywhere in the region from anywhere in the region which a signed relative *offset* would not allow.

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

31 26	25 0
JALX 011101	instr_index
6	26

Format: `JALX target` **MIPS64 with MIPS16e**

Purpose:

To execute a procedure call within the current 256 MB-aligned region and change the ISA Mode from 32-bit MIPS to MIPS16e.

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call. The value stored in GPR 31 bit 0 reflects the current value of the *ISA Mode* bit.

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

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

Restrictions:

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

Operation:

```
I:      GPR[31]← PC + 8
I+1:    PC ← PC_GPRLEN..28 || instr_index || 0^2
        ISAMode ← (not ISAMode)
```

Exceptions:

None

Programming Notes:

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

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Jump Register Through Register ra — JR

15–11	10–8	7	6	5	4–0
RR 11101	000	nd 0	l 0	ra 1	J(AL)R(C) 00000
5	3	1	1	1	5

Format: `JR ra` **MIPS16e**

Purpose:

To execute a branch to the instruction address in the return address register.

Description: `PC ← GPR[ra]`

The program unconditionally jumps to the address specified in GPR 31, with a delay of one instruction. The instruction sets the *ISA Mode* bit to the value in GPR 31 bit 0.

Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

The opcode and function field describe a general jump-thru-register operation, with the *nd* (no delay slot), *l* (link), and *ra* (source register is ra) fields as variables. The individual instructions, JALR, JR, JALRC, and JRC have specific values for these variables.

Restrictions:

The effective target address in GPR 31 must be naturally-aligned. If bit 0 is zero and bit 1 is one, then an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

An extended instruction should not be placed in a jump delay slot, because this causes one-half of an instruction to be executed.

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

Operation:

$$\text{I+1:}\quad PC \leftarrow GPR[31]_{GPRLEN-1..1} \,||\, 0$$
$$ISAMode \leftarrow GPR[31]_0$$

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Jump Register Through MIPS16e GPR **JR**

15–11	10–8	7	6	5	4–0
RR 11101	rx	nd 0	l 0	ra 0	J(AL)R(C) 00000
5	3	1	1	1	5

Format: `JR rx` **MIPS16e**

Purpose:

To execute a branch to an instruction address in a register.

Description: `PC ← GPR[rx]`

The program unconditionally jumps to the address specified in GPR *rx*, with a delay of one instruction. The instruction sets the *ISA Mode* bit to the value in GPR *rx* bit 0.

Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

The opcode and function field describe a general jump-thru-register operation, with the *nd* (no delay slot), *l* (link), and *ra* (source register is ra) fields as variables. The individual instructions, JALR, JR, JALRC, and JRC have specific values for these variables.

Restrictions:

The effective target address in GPR *rx* must be naturally aligned. If bit 0 is zero and bit 1 is one, then an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

An extended instruction should not be placed in a jump delay slot, because this causes one-half of an instruction to be executed.

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

Operation:

```
I+1:    PC ← GPR[Xlat(rx)]GPRLEN-1..1 || 0
        ISAMode ← GPR[Xlat(rx)]0
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

15 ... 11	10 ... 8	7	6	5	4 ... 0
RR	000	nd	l	ra	J(AL)R(C)
11101		1	0	1	00000
5	3	1	1	1	5

Format: `JRC ra` **MIPS16e**

Purpose:

To execute a branch to the instruction address in the return address register.

Description: `PC ← GPR[ra]`

The program unconditionally jumps to the address specified in GPR 31, with no delay slot instruction. The instruction sets the *ISA Mode* bit to the value in GPR 31 bit 0.

Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

The opcode and function field describe a general jump-thru-register operation, with the *nd* (no delay slot), *l* (link), and *ra* (source register is ra) fields as variables. The individual instructions, JALR, JR, JALRC, and JRC have specific values for these variables.

Restrictions:

The effective target address in GPR 31 must be naturally-aligned. If bit 0 is zero and bit 1 is one, then an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

Operation:

```
I:      PC ← GPR[31]GPRLEN-1..1 || 0
        ISAMode ← GPR[31]0
```

Exceptions:

None.

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

15–11	10–8	7	6	5	4–0
RR 11101	rx	nd 1	l 0	ra 0	J(AL)R(C) 00000
5	3	1	1	1	5

Format: `JRC rx` **MIPS16e**

Purpose:

To execute a branch to an instruction address in a register

Description: `PC ← GPR[rx]`

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

Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

The opcode and function field describe a general jump-thru-register operation, with the *nd* (no delay slot), *l* (link), and *ra* (source register is ra) fields as variables. The individual instructions, JALR, JR, JALRC, and JRC have specific values for these variables.

Restrictions:

The effective target address in GPR *rx* must be naturally-aligned. If bit 0 is zero and bit 1 is one, then an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

Operation:

```
I:    PC ← GPR[Xlat(rx)]GPRLEN-1..1 || 0
      ISAMode ← GPR[Xlat(rx)]0
```

Exceptions:

None.

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Byte **LB**

15 — 11	10 — 8	7 — 5	4 — 0
LB 10000	rx	ry	offset
5	3	3	5

Format: `LB ry, offset(rx)` **MIPS16e**

Purpose:

To load a byte from memory as a signed value.

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 5-bit *offset* is zero-extended, then added to the contents of GPR *rx* 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 *ry*.

Restrictions:

None

Operation:

```
vAddr  ← zero_extend(offset) + GPR[Xlat(rx)]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian^3)
memdoubleword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte   ← vAddr2..0 xor BigEndianCPU^3
GPR[Xlat(ry)]← sign_extend(memdoubleword7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Byte (Extended) **LB**

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	LB 10000	rx	ry	offset 4:0
5	6	5	5	3	3	5

Format: `LB ry, offset(rx)` **MIPS16e**

Purpose:

To load a byte from memory as a signed value.

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 16-bit *offset* is sign-extended, then added to the contents of GPR *rx* 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 *ry*.

Restrictions:

None

Operation:

```
vAddr  ← sign_extend(offset) + GPR[Xlat(rx)]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
memdoubleword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte   ← vAddr2..0 xor BigEndianCPU3
GPR[Xlat(ry)]← sign_extend(memdoubleword7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Byte Unsigned **LBU**

15 – 11	10 – 8	7 – 5	4 – 0
LBU 10100	rx	ry	offset
5	3	3	5

Format: `LBU ry, offset(rx)` **MIPS16e**

Purpose:

To load a byte from memory as an unsigned value

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 5-bit *offset* is zero-extended, then added to the contents of GPR *rx* 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 *ry*.

Restrictions:

None

Operation:

```
vAddr  ← zero_extend(offset) + GPR[Xlat(rx)]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_{PSIZE-1..3} || (pAddr_{2..0} xor ReverseEndian^3)
memdoubleword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte   ← vAddr_{2..0} xor BigEndianCPU^3
GPR[Xlat(ry)]← zero_extend(memdoubleword_{7+8*byte..8*byte})
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Byte Unsigned (Extended) LBU

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	LBU 10100	rx	ry	offset 4:0
5	6	5	5	3	3	5

Format: `LBU ry, offset(rx)` **MIPS16e**

Purpose:

To load a byte from memory as an unsigned value

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 16-bit *offset* is sign-extended, then added to the contents of GPR *rx* 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 *ry*.

Restrictions:

None

Operation:

```
vAddr  ← sign_extend(offset) + GPR[Xlat(rx)]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_{PSIZE-1..3} || (pAddr_{2..0} xor ReverseEndian^3)
memdoubleword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte   ← vAddr_{2..0} xor BigEndianCPU^3
GPR[Xlat(ry)]← zero_extend(memdoubleword_{7+8*byte..8*byte})
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Doubleword **LD**

15–11	10–8	7–5	4–0
LD 00111	rx	ry	offset
5	3	3	5

Format: `LD ry, offset(rx)` **MIPS16e (64-bit only)**

Purpose:

To load a doubleword from memory.

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 5-bit *offset* is shifted left 3 bits, zero-extended to 64 bits, then added to the contents of GPR *rx* to form the effective address. The contents of the 64-bit doubleword at the memory location specified by the effective address are loaded into GPR *ry*.

Restrictions:

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr          ← zero_extend(offset || 0^3) + GPR[Xlat(rx)]
if vAddr_2..0 ≠ 0^3 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[Xlat(ry)]← memdoubleword
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Doubleword (Extended) **LD**

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	LD 00111	rx	ry	offset 4:0
5	6	5	5	3	3	5

Format: `LD ry, offset(rx)` **MIPS16e (64-bit only)**

Purpose:

To load a doubleword from memory.

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 16-bit *offset* is sign-extended to 64 bits and then added to the contents of GPR *rx* to form the effective address. The contents of the 64-bit doubleword at the memory location specified by the effective address are loaded into GPR *ry*.

Restrictions:

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr          ← sign_extend(offset) + GPR[Xlat(rx)]
if vAddr2..0 ≠ 0^3 then
    SignalException(AddressError)
endif
(pAddr, CCA)  ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[Xlat(ry)] ← memdoubleword
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Doubleword (PC-Relative) **LD**

15 — 11	10 — 8	7 — 5	4 — 0
I64 11111	LDPC 100	ry	offset
5	3	3	5

Format: `LD ry, offset(pc)` **MIPS16e (64-bit only)**

Purpose:

To load a PC-relative doubleword from memory.

Description: `GPR[ry] ← memory[PC + offset]`

The 5-bit *offset* is shifted left 3 bits, zero-extended to 64 bits, and added either to the address of the LD instruction or to the address of the jump instruction in whose delay slot the LD is executed. The 3 lower bits of this result are cleared to form the effective address. The contents of the 64-bit doubleword at the memory location specified by the effective address are loaded into GPR *ry*.

Restrictions:

Operation:

```
I-1:  base_pc ← PC
I:    if not (JumpDelaySlot(PC)) then
          base_pc ← PC
      endif
vAddr ← (base_pc_{GPRLEN-1..3} + zero_extend(offset)) || 0^3
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[Xlat(ry)]← memdoubleword
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Reserved Instruction

Programming Note

For the purposes of watchpoints (provided by the CP0 *WatchHi* and *WatchLo* registers) and EJTAG breakpoints, the PC-relative reference is considered to be a data, rather than an instruction reference. That is, the watchpoint or breakpoint is triggered only if enabled for data references.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Doubleword (PC-Relative, Extended) **LD**

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	I64 11111	LDPC 100	ry	offset 4:0
5	6	5	5	3	3	5

Format: `LD ry, offset(pc)` **MIPS16e (64-bit only)**

Purpose:

To load a PC-relative doubleword from memory.

Description: `GPR[ry] ← memory[PC + offset]`

The 16-bit *offset* is sign-extended and added to the address of the LD instruction; this forms the effective address. Before the addition, the 3 lower bits of the instruction address are cleared. The contents of the 64-bit doubleword at the memory location specified by the effective address are loaded into GPR *ry*.

Restrictions:

A PC-relative, extended LD may not be placed in the delay slot of a jump instruction.

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← (PC_GPRLEN-1..3 || 0^3) + sign_extend(offset)
if vAddr_2..0 ≠ 0^3 then
SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[Xlat(ry)]← memdoubleword
```

Exceptions:

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

Programming Note

For the purposes of watchpoints (provided by the CP0 *WatchHi* and *WatchLo* registers) and EJTAG breakpoints, the PC-relative reference is considered to be a data, rather than an instruction reference. That is, the watchpoint or breakpoint is triggered only if enabled for data references.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Doubleword (SP-Relative) **LD**

15 ... 11	10 ... 8	7 ... 5	4 ... 0
I64 11111	LDSP 000	ry	offset
5	3	3	5

Format: `LD ry, offset(sp)` **MIPS16e (64-bit only)**

Purpose:

To load a doubleword from memory.

Description: `GPR[ry] ← memory[GPR[sp] + offset]`

The 5-bit *offset* is shifted left 3 bits, zero-extended to 64 bits, then added to the contents of GPR 29 to form the effective address. The contents of the 64-bit doubleword at the memory location specified by the effective address are loaded into GPR *ry*.

Restrictions:

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr         ← zero_extend(offset || 0^3) + GPR[29]
if vAddr_2..0 ≠ 0^3 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[Xlat(ry)]← memdoubleword
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Doubleword (SP-Relative, Extended) **LD**

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	I64 11111	LDSP 000	ry	offset 4:0
5	6	5	5	3	3	5

Format: `LD ry, offset(sp)` **MIPS16e (64-bit only)**

Purpose:

To load an SP-relative doubleword from memory.

Description: `GPR[ry] ← memory[GPR[sp] + offset]`

The 16-bit *offset* is sign-extended to 64 bits and then added to the contents of GPR 29 to form the effective address. The contents of the 64-bit doubleword at the memory location specified by the effective address are loaded into GPR *ry*.

Restrictions:

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr         ← sign_extend(offset) + GPR[29]
if vAddr2..0 ≠ 0^3 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[Xlat(ry)]← memdoubleword
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Halfword **LH**

15 … 11	10 … 8	7 … 5	4 … 0
LH 10001	rx	ry	offset
5	3	3	5

Format: `LH ry, offset(rx)` **MIPS16e**

Purpose:

To load a halfword from memory as a signed value.

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 5-bit *offset* is shifted left 1 bit, zero-extended, then added to the contents of GPR *rx* 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 *ry*.

Restrictions:

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

Operation:

```
vAddr  ← zero_extend(offset || 0) + GPR[Xlat(rx)]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian^2 || 0))
memdoubleword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte   ← vAddr2..0 xor (BigEndianCPU^2 || 0)
GPR[Xlat(ry)] ← sign_extend(memdoubleword15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Halfword (Extended) LH

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	LH 10001	rx	ry	offset 4:0
5	6	5	5	3	3	5

Format: `LH ry, offset(rx)` **MIPS16e**

Purpose:

To load a halfword from memory as a signed value.

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 16-bit *offset* is sign-extended and then added to the contents of GPR *rx* 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 *ry*.

Restrictions:

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

Operation:

```
vAddr  ← sign_extend(offset) + GPR[Xlat(rx)]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian^2 || 0))
memdoubleword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte   ← vAddr2..0 xor (BigEndianCPU^2 || 0)
GPR[Xlat(ry)] ← sign_extend(memdoubleword15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Halfword Unsigned — LHU

15–11	10–8	7–5	4–0
LHU 10101	rx	ry	offset
5	3	3	5

Format: `LHU ry, offset(rx)` **MIPS16e**

Purpose:

To load a halfword from memory as an unsigned value.

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 5-bit *offset* is shifted left 1 bit, zero-extended, then added to the contents of GPR *rx* 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 *ry*.

Restrictions:

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

Operation:

```
vAddr  ← zero_extend(offset || 0) + GPR[Xlat(rx)]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian2 || 0))
memdoubleword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte   ← vAddr2..0 xor (BigEndianCPU2 || 0)
GPR[Xlat(ry)] ← zero_extend(memdoubleword15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Halfword Unsigned (Extended) — LHU

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	LHU 10101	rx	ry	offset 4:0
5	6	5	5	3	3	5

Format: `LHU ry, offset(rx)` **MIPS16e**

Purpose:

To load a halfword from memory as an unsigned value.

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 16-bit *offset* is sign-extended and then added to the contents of GPR *rx* 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 *ry*.

Restrictions:

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

Operation:

```
vAddr  ← sign_extend(offset) + GPR[Xlat(rx)]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian2 || 0))
memdoubleword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte   ← vAddr2..0 xor (BigEndianCPU2 || 0)
GPR[Xlat(ry)] ← zero_extend(memdoubleword15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Immediate — LI

15–11	10–8	7–0
LI 01101	rx	immediate
5	3	8

Format: `LI rx, immediate` **MIPS16e**

Purpose:

To load a constant into a GPR.

Description: `GPR[rx] ← immediate`

The 8-bit *immediate* is zero-extended and then loaded into GPR *rx*.

Restrictions:

None

Operation:

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

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Immediate (Extended) **LI**

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	immediate 10:5	immediate 15:11	LI 01101	rx	0 000	iummediate 4:0
5	6	5	5	3	3	5

Format: `LI rx, immediate` **MIPS16e**

Purpose:

To load a constant into a GPR.

Description: `GPR[rx] ← immediate`

The 16-bit *immediate* is zero-extended and then loaded into GPR *rx*.

Restrictions:

None

Operation:

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

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Word **LW**

15 … 11	10 … 8	7 … 5	4 … 0
LW 10011	rx	ry	offset
5	3	3	5

Format: `LW ry, offset(rx)` **MIPS16e**

Purpose:

To load a word from memory as a signed value.

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 5-bit *offset* is shifted left 2 bits, zero-extended, then added to the contents of GPR *rx* to form the effective address. The contents of the word at the memory location specified by the effective address are loaded into GPR *ry.*

Restrictions:

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

Operation:

```
vAddr  ← zero_extend(offset || 0^2) + GPR[Xlat(rx)]
if vAddr_1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor (ReverseEndian || 0^2))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr_2..0 xor (BigEndianCPU || 0^2)
GPR[Xlat(ry)]← sign_extend(memdoubleword_31+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Word (Extended) — LW

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	LW 10011	rx	ry	offset 4:0
5	6	5	5	3	3	5

Format: `LW ry, offset(rx)` **MIPS16e**

Purpose:

To load a word from memory as a signed value.

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

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

Restrictions:

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

Operation:

```
vAddr  ← sign_extend(offset) + GPR[Xlat(rx)]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr2..0 xor (BigEndianCPU || 0^2)
GPR[Xlat(ry)]← sign_extend(memdoubleword31+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Word (PC-Relative) — LW

15 … 11	10 … 8	7 … 0
LWPC 10110	rx	offset
5	3	8

Format: `LW rx, offset(pc)` **MIPS16e**

Purpose:

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

Description: `GPR[rx] ← memory[PC + offset]`

The 8-bit *offset* is shifted left 2 bits, zero-extended, and added either to the address of the LW instruction or to the address of the jump instruction in whose delay slot the LW is executed. The 2 lower bits of this result are cleared to form the effective address. The contents of the 32-bit word at the memory location specified by the effective address are loaded into GPR *rx*.

Restrictions:

None

Operation:

```
I-1:  base_pc ← PC
I:    if not (JumpDelaySlot(PC)) then
         base_pc ← PC
      endif
      vAddr ← (base_pc_{GPRLEN-1..2} + zero_extend(offset)) || 0^2
      (pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
      pAddr  ← pAddr_{PSIZE-1..3} || (pAddr_{2..0} xor (ReverseEndian || 0^2))
      memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
      byte   ← vAddr_{2..0} xor (BigEndianCPU || 0^2)
      GPR[Xlat(rx)]← sign_extend(memdoubleword_{31+8*byte..8*byte})
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error

Programming Note

For the purposes of watchpoints (provided by the CP0 *WatchHi* and *WatchLo* registers) and EJTAG breakpoints, the PC-relative reference is considered to be a data, rather than an instruction reference. That is, the watchpoint or breakpoint is triggered only if enabled for data references.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Word (PC-Relative, Extended) LW

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	LWPC 10110	rx	0 000	offset 4:0
5	6	5	5	3	3	5

Format: `LW rx, offset(pc)` **MIPS16e**

Purpose:

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

Description: `GPR[rx] ← memory[PC + offset]`

The 16-bit *offset* is sign-extended and added to the address of the LW instruction; this forms the effective address. Before the addition, the 2 lower bits of the instruction address are cleared. The contents of the 32-bit word at the memory location specified by the effective address are loaded into GPR *rx*.

Restrictions:

A PC-relative, extended LW may not be placed in the delay slot of a jump instruction.

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 ← (PC_GPRLEN-1..2 || 02) + sign_extend(offset)
if vAddr_1..0 ≠ 0^2 then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor (ReverseEndian || 0^2))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr_2..0 xor (BigEndianCPU || 0^2)
GPR[Xlat(rx)]← sign_extend(memdoubleword_31+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Programming Note

For the purposes of watchpoints (provided by the CP0 *WatchHi* and *WatchLo* registers) and EJTAG breakpoints, the PC-relative reference is considered to be a data, rather than an instruction reference. That is, the watchpoint or breakpoint is triggered only if enabled for data references.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Word (SP-Relative) **LW**

15 — 11	10 — 8	7 — 0
LWSP 10010	rx	offset
5	3	8

Format: `LW rx, offset(sp)` **MIPS16e**

Purpose:

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

Description: `GPR[rx] ← memory[GPR[sp] + offset]`

The 8-bit *offset* is shifted left 2 bits, zero-extended, then added to the contents of GPR 29 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:

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

Operation:

```
vAddr  ← zero_extend(offset || 0^2) + GPR[29]
if vAddr_{1..0} ≠ 0^2 then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_{PSIZE-1..3} || (pAddr_{2..0} xor (ReverseEndian || 0^2))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr_{2..0} xor (BigEndianCPU || 0^2)
GPR[Xlat(ry)]← sign_extend(memdoubleword_{31+8*byte..8*byte})
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Word (SP-Relative, Extended) **LW**

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	LWSP 10010	rx	0 000	offset 4:0
5	6	5	5	3	3	5

Format: `LW rx, offset(sp)` **MIPS16e**

Purpose:

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

Description: `GPR[rx] ← memory[GPR[sp] + offset]`

The 16-bit *offset* is sign-extended and then added to the contents of GPR 29 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:

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

Operation:

```
vAddr  ← sign_extend(offset) + GPR[29]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr2..0 xor (BigEndianCPU || 0^2)
GPR[Xlat(ry)]← sign_extend(memdoubleword31+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Word Unsigned **LWU**

15 … 11	10 … 8	7 … 5	4 … 0
LWU 10111	rx	ry	offset
5	3	3	5

Format: `LWU ry, offset(rx)` **MIPS16e (64-bit only)**

Purpose:

To load a word from memory as an unsigned value.

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 5-bit *offset* is shifted left 2 bits, zero-extended to 64 bits, then added to the contents of GPR *rx* to form the effective address. The contents of the word at the memory location specified by the effective address are zero-extended and loaded into GPR *ry*.

Restrictions:

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

Operation:

```
vAddr  ← zero_extend(offset || 0^2) + GPR[Xlat(rx)]
if vAddr_1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor (ReverseEndian || 0^2))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr_2..0 xor (BigEndianCPU || 0^2)
GPR[Xlat(ry)]← 0^32 || memdoubleword_31+8*byte..8*byte
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Load Word Unsigned (Extended) — LWU

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	LWU 10111	rx	ry	offset 4:0
5	6	5	5	3	3	5

Format: `LWU ry, offset(rx)` **MIPS16e (64-bit only)**

Purpose:

To load a word from memory as an unsigned value.

Description: `GPR[ry] ← memory[GPR[rx] + offset]`

The 16-bit *offset* is sign-extended to 64 bits and then added to the contents of GPR *rx* to form the effective address. The contents of the word at the memory location specified by the effective address are zero-extended and loaded into GPR *ry*.

Restrictions:

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

Operation:

```
vAddr  ← sign_extend(offset) + GPR[Xlat(rx)]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr2..0 xor (BigEndianCPU || 0^2)
GPR[Xlat(ry)]← 0^32 || memdoubleword31+8*byte..8*byte
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Move From HI Register **MFHI**

15–11	10–8	7–5	4–0
RR 11101	rx	0 000	MFHI 10000
5	3	3	5

Format: `MFHI rx` **MIPS16e**

Purpose:

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

Description: `GPR[rx] ← HI`

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

Restrictions:

None

Operation:

```
GPR[Xlat(rx)] ← HI
```

Exceptions:

None

Historical Information:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Move From LO Register — MFLO

15 … 11	10 … 8	7 … 5	4 … 0
RR 11101	rx	0 000	MFLO 10010
5	3	3	5

Format: `MFLO rx` **MIPS16e**

Purpose:

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

Description: `GPR[rx] ← LO`

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

Restrictions:

None

Operation:

```
GPR[Xlat(rx)] ← LO
```

Exceptions:

None

Historical Information:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

15 — 11	10 — 8	7 — 5	4 — 3	2 — 0
I8 01100	MOV32R 101	r32 2:0	r32 4:3	rz
5	3	3	2	3

Format: `MOVE r32, rz` **MIPS16e**

Purpose:

To move the contents of a GPR to a GPR.

Description: `GPR[r32] ← GPR[rz]`

The contents of GPR *rz* are moved into GPR r32, and r32 can specify any one of the 32 GPRs.

Restrictions:

None

Operation:

```
GPR[r32] ← GPR[Xlat(rz)]
```

Exceptions:

None

Programming Notes:

move $0, $0, expressed as NOP, is the assembly idiom used to denote no operation.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Move **MOVE**

15 — 11	10 — 8	7 — 5	4 — 0
I8 01100	MOVR32 111	ry	r32
5	3	3	5

Format: `MOVE ry, r32` **MIPS16e**

Purpose:

To move the contents of a GPR to a GPR.

Description: `GPR[ry] ← GPR[r32]`

The contents of GPR r32 are moved into GPR *ry*, and r32 can specify any one of the 32 GPRs.

Restrictions:

None

Operation:

```
GPR[Xlat(ry)] ← GPR[r32]
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Multiply Word — MULT

15–11	10–8	7–5	4–0
RR 11101	rx	ry	MULT 11000
5	3	3	5

Format: `MULT rx, ry` **MIPS16e**

Purpose:

To multiply 32-bit signed integers.

Description: `(LO, HI) ← GPR[rx] × GPR[ry]`

The 32-bit word value in GPR *rx* is multiplied by the 32-bit value in GPR *ry*, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is sign-extended and placed into special register *LO*, and the high-order 32-bit word is sign-extended and placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

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

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
   UndefinedResult()
endif
prod  ← GPR[Xlat(rx)] * GPR[Xlat(ry)]
LO    ← sign_extend(prod31..0)
HI    ← sign_extend(prod63..32)
```

Exceptions:

None

Programming Notes:

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

Programs that require overflow detection must check for it explicitly.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Multiply Unsigned Word — MULTU

15–11	10–8	7–5	4–0
RR 11101	rx	ry	MULTU 11001
5	3	3	5

Format: `MULTU rx, ry` **MIPS16e**

Purpose:

To multiply 32-bit unsigned integers.

Description: `(LO, HI) ← GPR[rx] × GPR[ry]`

The 32-bit word value in GPR *rx* is multiplied by the 32-bit value in GPR *ry*, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit word of the result is sign-extended and placed into special register *LO*, and the high-order 32-bit word is sign-extended and placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

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

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UndefinedResult()
endif
prod ← (0 || GPR[Xlat(rx)]) * (0 || GPR[Xlat(ry)])
LO ← sign_extend(prod31..0)
HI ← sign_extend(prod63..32)
```

Exceptions:

None

Programming Notes:

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

Programs that require overflow detection must check for it explicitly.

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Negate **NEG**

15 — 11	10 — 8	7 — 5	4 — 0
RR 11101	rx	ry	NEG 01011
5	3	3	5

Format: `NEG rx, ry` **MIPS16e**

Purpose:

To negate an integer value.

Description: `GPR[rx] ← 0 - GPR[ry]`

The contents of GPR *ry* are subtracted from zero to form a 32-bit result. The result is placed in GPR *rx*.

Restrictions:

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

Operation:

```
if (NotWordValue(GPR[Xlat(ry)])) then
    UNPREDICTABLE
endif
temp ← 0 - GPR[Xlat(ry)]
GPR[Xlat(rx)] ← sign_extend(temp31..0)
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

No Operation **NOP**

15 — 11	10 — 8	7 — 5	4 — 3	2 — 0
I8 01100	MOV32R 101	0 000	0 00	0 000
5	3	3	2	3

Format: `NOP` **MIPS16e Assembly Idiom**

Purpose:

To perform no operation.

Description:

NOP is the assembly idiom used to denote no operation. The actual instruction is interpreted by the hardware as MOVE $0,$16.

Restrictions:

None

Operation:

```
None
```

Exceptions:

None

Programming Notes:

The 0x6500 instruction word, which represents MOVE $0,$16, is the preferred NOP for software to use to fill jump delay slots and to pad out alignment sequences.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Not NOT

15	11	10	8	7	5	4	0
RR 11101		rx		ry		NOT 01111	
5		3		3		5	

Format: `NOT rx, ry` **MIPS16e**

Purpose:

To complement an integer valu

Description: `GPR[rx] ← (NOT GPR[ry])`

The contents of GPR *ry* are bitwise-inverted and placed in GPR *rx*.

Restrictions:

None

Operation:

```
GPR[Xlat(rx)] ← (not GPR[Xlat(ry)])
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

15 ... 11	10 ... 8	7 ... 5	4 ... 0
RR 11101	rx	ry	OR 01101
5	3	3	5

Format: `OR rx, ry` **MIPS16e**

Purpose:

To do a bitwise logical OR.

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

The contents of GPR *ry* are combined with the contents of GPR *rx* in a bitwise logical OR operation. The result is placed in GPR *rx*.

Restrictions:

None

Operation:

```
GPR[Xlat(rx)] ← GPR[Xlat(rx)] or GPR[Xlat(ry)]
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

15 – 11	10 – 8	7	6	5	4	3 – 0
I8 01100	SVRS 100	s 0	ra	s0	s1	framesize
5	3	1	1	1	1	4

Format: `RESTORE {ra,}{s0/s1/s0-1,}{framesize} (All args are optional)` **MIPS16e**

Purpose:

To deallocate a stack frame before exit from a subroutine, restoring return address and static registers, and adjusting stack

Description: `GPR[ra] ← Stack and/or GPR[17]← Stack and/or GPR[16]← Stack,`
`sp ← sp + (framesize*8)`

Restore the ra and/or GPR 16 and/or GPR 17 (s0 and s1 in the MIPS ABI calling convention) registers from the stack if the corresponding *ra*, *s0*, or *s1* bits of the instruction are set, and adjust the stack pointer by 8 times the *framesize* value. Registers are loaded from the stack assuming higher numbered registers are stored at higher stack addresses. A *framesize* value of 0 is interpreted as a stack adjustment of 128. On a MIPS64 implementation, words are loaded from the stack, sign-extended and loaded into the corresponding GPR, using the equivalent of load word.

The opcode and function field describe a general save/restore operation, with the *s* fields as a variables. The individual instructions, RESTORE and SAVE have specific values for this variable.

Restrictions:

If either of the 2 least-significant bits of the stack pointer are not zero, and any of the *ra*, *s0*, or *s1* bits are set, then an Address Error exception will occur.

Operation:

```
if framesize = 0 then
    temp ← GPR[29] + 128
else
    temp ← GPR[29] + (0 || (framesize << 3))
endif
temp2 ← temp
if ra = 1 then
    temp ← temp − 4
    GPR[31] ← LoadStackWord(temp)
endif
if s1 = 1 then
    temp ← temp - 4
    GPR[17] ← LoadStackWord(temp)
endif
if s0 = 1 then
    temp ← temp − 4
    GPR[16] ← LoadStackWord(temp)
endif
GPR[29] ← temp2

LoadStackWord(vaddr)
    if vAddr_1..0 ≠ 0^2 then
        SignalException(AddressError)
    endif
    (pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
    pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor (ReverseEndian || 0^2))
```


Copyright © 2005 MIPS Technologies Inc. All rights reserved.

```
    memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
    byte   ← vAddr2..0 xor (BigEndianCPU || 0^2)
    LoadStackWord ← sign_extend(memdoubleword31+8*byte..8*byte)
enfunction LoadStackWord
```

Exceptions:

TLB refill, TLB invalid, Address error, Bus Error

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

31–27	26–24	23–20	19–16	15–11	10–8	7	6	5	4	3–0
EXTEND 11110	xsregs	framesize 7:4	aregs	I8 01100	SVRS 100	s 0	ra	s0	s1	framesize 3:0
5	3	4	4	5	3	1	1	1	1	4

Format: `RESTORE {ra,}{xsregs,}{aregs,}{framesize}(All arguments optional)` **MIPS16e**

Purpose:

To deallocate a stack frame before exit from a subroutine, restoring return address and static registers from an extended static register set, and adjusting the stack

Description: `GPR[ra] ← Stack and/or GPR[18-23,30] ← Stack and/or GPR[17] ← Stack and/or GPR[16] ← Stack and/or GPR[4-7] ← Stack, sp ← sp + (framesize * 8)`

Restore the ra register from the stack if the *ra* bit is set in the instruction. Restore from the stack the number of registers in the set GPR[18-23,30] indicated by the value of the *xsregs* field. Restore from the stack GPR 16 and/or GPR 17 (s0 and s1 in the MIPS ABI calling convention) from the stack if the corresponding *s0* and *s1* bits of the instruction are set, restore from the stack the number of registers in the range GPR[4-7] indicated by the *aregs* field, and adjust the stack pointer by 8 times the 8-bit concatenated *framesize* value. Registers are loaded from the stack assuming higher numbered registers are stored at higher stack addresses. On a MIPS64 implementation, words are loaded from the stack, sign-extended and loaded into the corresponding GPR, using the equivalent of load word.

Interpretation of the *aregs* Field

In the standard MIPS ABIs, GPR[4-7] are designated as argument passing registers, a0-a3. When they are so used, they must be saved on the stack at locations allocated by the caller of the routine being entered, but need not be restored on subroutine exit. In other MIPS16e calling sequences, however, it is possible that some of the registers GPR[4-7] need to be saved as static registers on the local stack instead of on the caller stack, and restored before return from the subroutine. The encoding used for the *aregs* field of an extended RESTORE instruction is the same as that used for the extended SAVE, but since argument registers can be ignored for the purposes of a RESTORE, only the registers treated as static need be handled. The following table shows the RESTORE encoding of the *aregs* field

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

***aregs* Encoding (binary)**	**Registers Restored as Static Registers**
0 0 0 0	None
0 0 0 1	GPR[7]
0 0 1 0	GPR[6], GPR[7]
0 0 1 1	GPR[5], GPR[6], GPR[7]
1 0 1 1	GPR[4], GPR[5], GPR[6]. GPR[7]
0 1 0 0	None
0 1 0 1	GPR[7]
0 1 1 0	GPR[6], GPR[7]
0 1 1 1	GPR[5], GPR[6], GPR[7]
1 0 0 0	None
1 0 0 1	GPR[7]
1 0 1 0	GPR[6], GPR[7]
1 1 0 0	None
1 1 0 1	GPR[7]
1 1 1 0	None
1 1 1 1	Reserved

Restrictions:

If either of the 2 least-significant bits of the stack pointer are not zero, and any of the *ra*, *s0*, *s1*, or *xsregs* fields are non-zero or the *aregs* field contains an encoding that implies a register load, then an Address Error exception will occur.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Operation:

```
temp ← GPR[29] + (0 ‖ (framesize << 3))
temp2 ← temp
if ra = 1 then
    temp ← temp – 4
    GPR[31] ← LoadStackWord(temp)
endif
if xsregs > 0 then
    if xsregs > 1 then
        if xsregs > 2 then
            if xsregs > 3 then
                if xsregs > 4 then
                    if xsregs > 5 then
                        if xsregs > 6 then
                            temp ← temp – 4
                            GPR[30] ← LoadStackWord(temp)
                        endif
                        temp ← temp – 4
                        GPR[23] ← LoadStackWord(temp)
                    endif
                    temp ← temp – 4
                    GPR[22] ← LoadStackWord(temp)
                endif
                temp ← temp ¯ 4
                GPR[21] ← LoadStackWord(temp)
            endif
            temp ← temp – 4
            GPR[20] ← LoadStackWord(temp)
        endif
        temp ← temp – 4
        GPR[19] ← LoadStackWord(temp)
    endif
    temp ← temp – 4
    GPR[18] ← LoadStackWord(temp)
endif
if s1 = 1 then
    temp ← temp – 4
    GPR[17] ← LoadStackWord(temp)
endif
if s0 = 1 then
    temp ← temp – 4
    GPR[16] ← LoadStackWord(temp)
endif
case aregs of
    0b0000 0b0100 0b1000 0b1100 0b1110: astatic ← 0
    0b0001 0b0101 0b1001 0b1101: astatic ← 1
    0b0010 0b0110 0b1010: astatic ← 2
    0b0011 0b0111: astatic ← 3
    0b1011: astatic ← 4
    otherwise: UNPREDICTABLE
endcase
```


Copyright © 2005 MIPS Technologies Inc. All rights reserved.

```
if astatic > 0 then
    temp ← temp − 4
    GPR[7] ← LoadStackWord(temp)
    if astatic > 1 then
        temp ← temp − 4
        GPR[6] ← LoadStackWord(temp)
        if astatic > 2 then
            temp ← temp − 4
            GPR[5] ← LoadStackWord(temp)
            if astatic > 3 then
                temp ← temp − 4
                GPR[4] ← LoadStackWord(temp)
            endif
        endif
    endif
endif
GPR[29] ← temp2

LoadStackWord(vaddr)
    if vAddr1..0 ≠ 0^2 then
        SignalException(AddressError)
    endif
    (pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
    pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
    memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
    byte   ← vAddr2..0 xor (BigEndianCPU || 0^2)
    LoadStackWord ← sign_extend(memdoubleword31+8*byte..8*byte)
enfunction LoadStackWord
```

Exceptions:

TLB refill, TLB invalid, Address error, Bus Error

Programming Notes:

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

Behavior of the processor is **UNPREDICTABLE** for Reserved values of *aregs*.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

15–11	10–8	7	6	5	4	3–0
I8 01100	SVRS 100	s 1	ra	s0	s1	framesize
5	3	1	1	1	1	4

Format: `SAVE {ra,}{s0/s1/s0-1,}{framesize} (All arguments are optional)` **MIPS16e**

Purpose:

To set up a stack frame on entry to a subroutine, saving return address and static registers, and adjusting stack

Description: `Stack ← GPR[ra] and/or Stack ← GPR[17] and/or Stack ← GPR[16],`
`sp ← sp - (framesize * 8)`

Save the *ra* and/or GPR 16 and/or GPR 17 (s0 and s1 in the MIPS ABI calling convention) on the stack if the corresponding *ra*, *s0*, and *s1* bits of the instruction are set, and adjust the stack pointer by 8 times the *framesize* value. Registers are stored with higher numbered registers at higher stack addresses. A *framesize* value of 0 is interpreted as a stack adjustment of 128. On a MIPS64 implementation, only the lower 32 bits of each GPR are saved, using the equivalent of store word.

The opcode and function field describe a general save/restore operation, with the *s* fields as a variables. The individual instructions, RESTORE and SAVE have specific values for this variable.

Restrictions:

If either of the 2 least-significant bits of the stack pointer are not zero, and any of the *ra*, *s0*, or *s1* bits are set, then an Address Error exception will occur.

Operation:

```
temp ← GPR[29]
if ra = 1 then
    temp ← temp − 4
    StoreStackWord(temp, GPR[31])
endif
if s1 = 1 then
    temp ← temp − 4
    StoreStackWord(temp, GPR[17])
endif
if s0 = 1 then
    temp ← temp − 4
    StoreStackWord(temp, GPR[16])
endif
if framesize = 0 then
    temp ← GPR[29] − 128
else
    temp ← GPR[29] − (0 || (framesize << 3))
endif
GPR[29] ← temp

StoreStackWord(vaddr, value)
    if vAddr_1..0 ≠ 0^2 then
        SignalException(AddressError)
    endif
    (pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
    pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor (ReverseEndian || 0^2))
    bytesel← vAddr_2..0 xor (BigEndianCPU || 0^2)
```


Copyright © 2005 MIPS Technologies Inc. All rights reserved.

```
    datadoubleword← value63-8*bytesel..0 || 0^(8*bytesel)
    StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
endfunction StoreStackWord
```

Exceptions:

TLB refill, TLB invalid, TLB modified, Address error, Bus Error

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

31–27	26–24	23–20	19–16	15–11	10–8	7	6	5	4	3–0
EXTEND 11110	xsregs	framesize 7:4	aregs	I8 01100	SVRS 100	s 1	ra	s0	s1	framesize 3:0
5	3	4	4	5	3	1	1	1	1	4

Format: `SAVE {ra,}{xsregs,}{aregs,}{framesize} (All arguments optional)` **MIPS16e**

Purpose:

To set up a stack frame on entry to a subroutine, saving return address, static, and argument registers, and adjusting the stack

Description: `Stack ← GPR[ra] and/or Stack ← GPR[18-23,30] and/or Stack ← GPR[17] and/or Stack ← GPR[16] and/or Stack ← GPR[4-7], sp ← sp - (framesize * 8)`

Save registers GPR[4-7] specified to be treated as incoming arguments by the *aregs* field. Save the ra register on the stack if the *ra* bit of the instruction is set. Save the number of registers in the set GPR[18-23, 30] indicated by the value of the *xsregs* field, and/or GPR 16 and/or GPR 17 (s0 and s1 in the MIPS ABI calling convention) on the stack if the corresponding *s0* and *s1* bits of the instruction are set. Save the number of registers in the range GPR[4-7] that are to be treated as static registers as indicated by the *aregs* field, and adjust the stack pointer by 8 times the 8-bit concatenated *framesize* value. Registers are stored with higher numbered registers at higher stack addresses. On a MIPS64 implementation, only the lower 32 bits of each GPR are saved, using the equivalent of store word.

Interpretation of the *aregs* Field

In the standard MIPS ABIs, GPR[4-7] are designated as argument passing registers, a0-a3. When they are so used, they must be saved on the stack at locations allocated by the caller of the routine being entered. In other MIPS16e calling sequences, however, it is possible that some of the registers GPR[4-7] will need to be saved as static registers on the local stack instead of on the caller stack. The encoding of the *aregs* field allows for 0-4 arguments, 0-4 statics, and for mixtures of the two. Registers are bound to arguments in ascending order, a0, a1, a2, and a3, and thus assigned to static values in the reverse order, GPR[7], GPR[6], GPR[5], and GPR[4]. The following table shows the encoding of the *aregs* field.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

aregs **Encoding (binary)**	**Registers Saved as Arguments**	**Registers Saved as Static Registers**
0 0 0 0	None	None
0 0 0 1	None	GPR[7]
0 0 1 0	None	GPR[6], GPR[7]
0 0 1 1	None	GPR[5], GPR[6], GPR[7]
1 0 1 1	None	GPR[4], GPR[5], GPR[6], GPR[7]
0 1 0 0	a0	None
0 1 0 1	a0	GPR[7]
0 1 1 0	a0	GPR[6], GPR[7]
0 1 1 1	a0	GPR[5], GPR[6], GPR[7]
1 0 0 0	a0, a1	None
1 0 0 1	a0, a1	GPR[7]
1 0 1 0	a0, a1	GPR[6], GPR[7]
1 1 0 0	a0, a1, a2	None
1 1 0 1	a0, a1, a2	GPR[7]
1 1 1 0	a0, a1, a2, a3	None
1 1 1 1	Reserved	Reserved

Restrictions:

If either of the 2 least-significant bits of the stack pointer are not zero, and any of the *ra*, *s0*, *s1*, or *xsregs* fields are non-zero or the *aregs* field contains an value that implies a register store, then an Address Error exception will occur.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Operation:

```
temp ← GPR[29]
temp2 ← GPR[29]
case aregs of
    0b0000 0b0001 0b0010 0b0011 0b1011: args ← 0
    0b0100 0b0101 0b0110 0b0111: args ← 1
    0b1000 0b1001 0b1010: args ← 2
    0b1100 0b1101: args ← 3
    0b1110: args ← 4
    otherwise: UNPREDICTABLE
endcase
if args > 0 then
    StoreStackWord(temp, GPR[4])
    if args > 1 then
        StoreStackWord(temp + 4, GPR[5])
        if args > 2 then
            StoreStackWord(temp + 8, GPR[6])
            if args > 3 then
                StoreStackWord(temp + 12, GPR[7])
            endif
        endif
    endif
endif
if ra = 1 then
    temp ← temp − 4
    StoreStackWord(temp, GPR[31])
endif
if xsregs > 0 then
    if xsregs > 1 then
        if xsregs > 2 then
            if xsregs > 3 then
                if xsregs > 4 then
                    if xsregs > 5 then
                        if xsregs > 6 then
                            temp ← temp − 4
                            StoreStackWord(temp, GPR[30])
                        endif
                        temp ← temp − 4
                        StoreStackWord(temp, GPR[23])
                    endif
                    temp ← temp − 4
                    StoreStackWord(temp, GPR[22])
                endif
                temp ← temp − 4
                StoreStackWord(temp, GPR[21])
            endif
            temp ← temp − 4
            StoreStackWord(temp, GPR[20])
        endif
        temp ← temp − 4
        StoreStackWord(temp, GPR[19])
    endif
    temp ← temp − 4
    StoreStackWord(temp, GPR[18])
endif
```


Copyright © 2005 MIPS Technologies Inc. All rights reserved.

```
if s1 = 1 then
    temp ← temp − 4
    StoreStackWord(temp, GPR[17])
endif
if s0 = 1 then
    temp ← temp − 4
    StoreStackWord(temp, GPR[16])
endif
case aregs of
    0b0000 0b0100 0b1000 0b1100 0b1110: astatic ← 0
    0b0001 0b0101 0b1001 0b1101: astatic ← 1
    0b0010 0b0110 0b1010: astatic ← 2
    0b0011 0b0111: astatic ← 3
    0b1011: astatic ← 4
    otherwise: UNPREDICTABLE
endcase
if astatic > 0 then
    temp ← temp − 4
    StoreStackWord(temp, GPR[7])
    if astatic > 1 then
        temp ← temp − 4
        StoreStackWord(temp, GPR[6])
        if astatic > 2 then
            temp ← temp − 4
            StoreStackWord(temp, GPR[5])
            if astatic > 3 then
                temp ← temp − 4
                StoreStackWord(temp, GPR[4])
            endif
        endif
    endif
endif
temp ← temp2 − (0 || (framesize << 3))
GPR[29] ← temp

StoreStackWord(vaddr, value)
    if vAddr1..0 ≠ 0^2 then
        SignalException(AddressError)
    endif
    (pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
    pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
    bytesel← vAddr2..0 xor (BigEndianCPU || 0^2)
    datadoubleword← value63-8*bytesel..0 || 0^(8*bytesel)
    StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
endfunction StoreStackWord
```

Exceptions:

TLB refill, TLB invalid, TLB modified, Address error, Bus Error

Programming Notes:

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

Behavior of the processor is **UNPREDICTABLE** for Reserved values of *aregs*.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Byte SB

15 … 11	10 … 8	7 … 5	4 … 0
SB 11000	rx	ry	offset
5	3	3	5

Format: `SB ry, offset(rx)` **MIPS16e**

Purpose:

To store a byte to memory.

Description: `memory[GPR[rx] + offset] ← GPR[ry]`

The 5-bit *offset* is zero-extended, then added to the contents of GPR *rx* to form the effective address. The least-significant byte of GPR *ry* is stored at the effective address.

Restrictions:

None

Operation:

```
vAddr       ← zero_extend(offset) + GPR[Xlat(rx)]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr       ← pAddr_{PSIZE-1..3} || (pAddr_{2..0} xor ReverseEndian^3)
bytesel     ← vAddr_{2..0} xor BigEndianCPU^3
datadoubleword← GPR[rt]_{63-8*bytesel..0} || 0^{8*bytesel}
StoreMemory (CCA, BYTE, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Byte (Extended) SB

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	SB 11000	rx	ry	offset 4:0
5	6	5	5	3	3	5

Format: `SB ry, offset(rx)` **MIPS16e**

Purpose:

To store a byte to memory.

Description: `memory[GPR[rx] + offset] ← GPR[ry]`

The 16-bit *offset* is sign-extended and then added to the contents of GPR *rx* to form the effective address. The least-significant byte of GPR *ry* is stored at the effective address.

Restrictions:

None

Operation:

```
vAddr      ← sign_extend(offset) + GPR[Xlat(rx)]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr      ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor ReverseEndian^3)
bytesel    ← vAddr_2..0 xor BigEndianCPU^3
datadoubleword← GPR[rt]_63-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, BYTE, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Sign-Extend Byte **SEB**

15–11	10–8	7–5	4–0
RR 11101	rx	SEB 100	CNVT 10001
5	3	3	5

Format: `SEB rx` **MIPS16e**

Purpose:

Sign-extend least significant byte in register rx.

Description: `GPR[rx] ← sign_extend(GPR[rx]`$_{7..0}$`)`

The least significant byte of rx is sign-extended and the value written back to rx.

Restrictions:

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

Operation:

```
if NotWordValue(GPR[Xlat(rx)]) then
    UNPREDICTABLE
endif
temp ← GPR[Xlat(rx)]
GPR[Xlat(rx)] ←sign_extend(temp7..0)
```

Exceptions:

None

Programming Notes:

None.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Sign-Extend Halfword SEH

15 — 11	10 — 8	7 — 5	4 — 0
RR 11101	rx	SEH 101	CNVT 10001
5	3	3	5

Format: `SEH rx` **MIPS16e**

Purpose:

Sign-extend least significant word in register rx.

Description: `GPR[rx] ← sign_extend(GPR[rx]`$_{15..0}$`);`

The least significant halfword of rx is sign-extended and the value written back to rx.

Restrictions:

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

Operation:

```
if NotWordValue(GPR[Xlat(rx)]) then
    UNPREDICTABLE
endif
temp ← GPR[Xlat(rx)]
GPR[Xlat(rx)] ←sign_extend(temp15..0)
```

Exceptions:

None

Programming Notes:

None.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Sign-Extend Word **SEW**

15–11	10–8	7–5	4–0
RR 11101	rx	SEW 110	CNVT 10001
5	3	3	5

Format: `SEW rx` **MIPS16e (64-bit only)**

Purpose:

Sign-extend least significant word in register rx.

Description: `GPR[rx] ← sign_extend(GPR[rx]`$_{31..0}$`);`

The least significant word of rx is sign-extended and the value written back to rx.

Restrictions:

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

Operation:

```
if NotWordValue(GPR[Xlat(rx)]) then
    UNPREDICTABLE
endif
temp ← GPR[Xlat(rx)]
GPR[Xlat(rx)] ←sign_extend(temp31..0)
```

Exceptions:

Reserved Instruction

Programming Notes:

None.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Doubleword **SD**

15 … 11	10 … 8	7 … 5	4 … 0
SD 01111	rx	ry	offset
5	3	3	5

Format: `SD ry, offset(rx)` **MIPS16e (64-bit only)**

Purpose:

To store a doubleword to memory.

Description: `memory[GPR[rx] + offset] ← GPR[ry]`

The 5-bit *offset* is shifted left 3 bits, zero-extended to 64 bits, and then added to the contents of GPR *rx* to form the effective address. The 64-bit contents of GPR *ry* are stored at the effective address.

Restrictions:

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← zero_extend(offset|| 0^3) + GPR[Xlat(rx)]
if vAddr_2..0 ≠ 0^3 then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
datadoubleword← GPR[Xlat(ry)]
StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Doubleword (Extended) SD

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	SD 01111	rx	ry	offset 4:0
5	6	5	5	3	3	5

Format: `SD ry, offset(rx)` **MIPS16e (64-bit only)**

Purpose:

To store a doubleword to memory.

Description: `memory[GPR[rx] + offset] ← GPR[ry]`

The 16-bit *offset* is sign-extended to 64 bits and then added to the contents of GPR *rx* to form the effective address. The 64-bit contents of GPR *ry* are stored at the effective address.

Restrictions:

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr  ← sign_extend(offset) + GPR[Xlat(rx)]
if vAddr2..0 ≠ 0^3 then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
datadoubleword← GPR[Xlat(ry)]
StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Doubleword ry (SP-Relative) **SD**

15 – 11	10 – 8	7 – 5	4 – 0
I64 11111	SDSP 001	ry	offset
5	3	3	5

Format: `SD ry, offset(sp)` **MIPS16e (64-bit only)**

Purpose:

To store an SP-relative doubleword to memory.

Description: `memory[GPR[sp] + offset] ← GPR[ry]`

The 5-bit *offset* is shifted left 3 bits, zero-extended to 64 bits, and then added to the contents of GPR 29 to form the effective address. The 64-bit contents of GPR *ry* are stored at the effective address.

Restrictions:

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr  ← zero_extend(offset || 0^3) + GPR[29]
if vAddr_2..0 ≠ 0^3 then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
datadoubleword← GPR[Xlat(ry)]
StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Doubleword ry (SP-Relative, Extended) **SD**

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	I64 11111	SDSP 001	ry	offset 4:0
5	6	5	5	3	3	5

Format: `SD ry, offset(sp)` **MIPS16e (64-bit only)**

Purpose:

To store an SP-relative doubleword to memory

Description: `memory[GPR[sp] + offset] ← GPR[ry]`

The 16-bit *offset* is sign-extended to 64 bits and then added to the contents of GPR 29 to form the effective address. The 64-bit contents of GPR *ry* are stored at the effective address.

Restrictions:

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr  ← sign_extend(offset) + GPR[29]
if vAddr2..0 ≠ 0^3 then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
datadoubleword← GPR[Xlat(ry)]
StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Doubleword ra (SP-Relative) — SD

15 — 11	10 — 8	7 — 0
I64 11111	SDRASP 010	offset
5	3	8

Format: `SD ra, offset(sp)` **MIPS16e (64-bit only)**

Purpose:

To store register *ra* SP-relative to memory.

Description: `memory[sp + offset] ← ra`

The 8-bit *offset* is shifted left 3 bits, zero-extended to 64 bits, and then added to the contents of GPR 29 to form the effective address. The 64-bit contents of GPR 31 are stored at the effective address.

Restrictions:

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation: 64-bit processors

```
vAddr ← GPR[29] + zero_extend(offset || 0^3)
if(vAddr_{2..0}) ≠ 0^3 then
   SignalException(AddressError)
endif
(pAddr,uncached) ← AddressTranslation(vAddr,DATA,STORE)
datadouble ← GPR[31]
StoreMemory(uncached,DOUBLEWORD,datadouble,pAddr,vAddr,DATA)
```

Exceptions:

Address Error, Reserved Instruction

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Doubleword ra (SP-Relative, Extended) — SD

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	I64 11111	SDRASP 010	ra	offset 4:0
5	6	5	5	3	3	5

Format: `SD ra, offset(sp)` **MIPS16e (64-bit only)**

Purpose:

To store register *ra* SP-relative to memory.

Description: `memory[sp + offset] ← ra`

The 16-bit *offset* is sign-extended to 64 bits and then added to the contents of GPR 29 to form the effective address. The 64-bit contents of GPR 31 are stored at the effective address.

Restrictions:

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation: 64-bit processors

```
vAddr ← GPR[29] + sign_extend(offset)
if(vAddr2..0) ≠ 0^3 then
   SignalException(AddressError)
endif
(pAddr,uncached) ← AddressTranslation(vAddr,DATA,STORE)
datadouble ← GPR[31]
StoreMemory(uncached,DOUBLEWORD,datadouble,pAddr,vAddr,DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

15–11	10–5	4–0
RR 11101	code	SDBBP 00001
5	6	5

Format: SDBBP code **EJTAG**

Purpose:

To cause a debug breakpoint exception

Description:

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

Restrictions:

Operation:

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

Exceptions:

Debug Breakpoint Exception
Debug Mode Breakpoint Exception

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Halfword **SH**

15–11	10–8	7–5	4–0
SH 11001	rx	ry	offset
5	3	3	5

Format: `SH ry, offset(rx)` **MIPS16e**

Purpose:

To store a halfword to memory.

Description: `memory[GPR[rxGPR[ + offset] ← GPR[ry]`

The 5-bit *offset* is shifted left 1 bit, zero-extended, and then added to the contents of GPR *rx* to form the effective address. The least-significant halfword of GPR *ry* is stored at the effective address.

Restrictions:

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

Operation:

```
vAddr  ← zero_extend(offset || 0) + GPR[Xlat(rx)]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddrPSIZE-1..3 || (pAddr12..0 xor (ReverseEndian2 || 0))
bytesel← vAddr12..0 xor (BigEndianCPU2 || 0)
datadoubleword← GPR[Xlat(ry)]63-8*bytesel..0 || 08*bytesel
StoreMemory (CCA, HALFWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Halfword (Extended) SH

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	SH 11001	rx	ry	offset 4:0
5	6	5	5	3	3	5

Format: `SH ry, offset(rx)` **MIPS16e**

Purpose:

To store a halfword to memory.

Description: `memory[GPR[rx] + offset] ← GPR[ry]`

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

Restrictions:

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

Operation:

```
vAddr  ← sign_extend(offset) + GPR[Xlat(rx)]
if vAddr_0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr1_2..0 xor (ReverseEndian^2 || 0))
bytesel← vAddr1_2..0 xor (BigEndianCPU^2 || 0)
datadoubleword← GPR[Xlat(ry)]_63-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, HALFWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Shift Word Left Logical SLL

15–11	10–8	7–5	4–2	1–0
SHIFT 00110	rx	ry	sa	SLL 00
5	3	3	3	2

Format: `SLL rx, ry, sa` **MIPS16e**

Purpose:

To execute a left-shift of a word by a fixed number of bits—1 to 8 bits.

Description: `GPR[rx] ← GPR[ry] << sa`

The 32-bit contents of GPR *ry* are shifted left, and zeros are inserted into the emptied low-order bits. The 3-bit *sa* field specifies the shift amount. A shift amount of 0 is interpreted as a shift amount of 8. The result is sign-extended and placed into GPR *rx*.

Restrictions:

None

Operation:

```
if sa = 0^3 then
    s ← 8
else
    s ← 0^2 || sa
endif
temp ← GPR[Xlat(ry)](31-s)..0 || 0^s
GPR[Xlat(rx)] ← sign_extend(temp31..0)
```

Exceptions:

None

Programming Notes:

Unlike nearly all other word operations, the SLL input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign-extended into a 64-bit destination register.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Shift Word Left Logical (Extended) — SLL

31–27	26–22	21–16	15–11	10–8	7–5	4–2	1–0
EXTEND 11110	sa4:0	0 000000	SHIFT 00110	rx	ry	0 000	SLL 00
5	5	6	5	3	3	3	2

Format: `SLL rx, ry, sa` **MIPS16e**

Purpose:

To execute a left-shift of a word by a fixed number of bits—0 to 31 bits.

Description: `GPR[rx] ← GPR[ry] << sa`

The 32-bit contents of GPR *ry* are shifted left, and zeros are inserted into the emptied low-order bits. The 5-bit *sa* field specifies the shift amount. The result is sign-extended and placed into GPR *rx*.

Restrictions:

None

Operation:

```
s ← sa
temp ← GPR[Xlat(ry)](31-s)..0 || 0^s
GPR[Xlat(rx)] ← sign_extend(temp31..0)
```

Exceptions:

None

Programming Notes:

Unlike nearly all other word operations, the SLL input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign-extended into a 64-bit destination register.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Shift Word Left Logical Variable **SLLV**

15 — 11	10 — 8	7 — 5	4 — 0
RR 11101	rx	ry	SLLV 00100
5	3	3	5

Format: `SLLV ry, rx` **MIPS16e**

Purpose:

To execute a left-shift of a word by a variable number of bits.

Description: `GPR[ry] ← GPR[ry] << GPR[rx]`

The 32-bit contents of GPR *ry* are shifted left, and zeros are inserted into the emptied low-order bits; the result word is sign-extended and and placed back in GPR *ry.* The 5 low-order bits of GPR *rx* specify the shift amount.

Restrictions:

None

Operation:

$$\texttt{s} \leftarrow \texttt{GPR[Xlat(rx)]}_{4..0}$$
$$\texttt{temp} \leftarrow \texttt{GPR[Xlat(ry)]}_{(31-s)..0} \;||\; 0^{s}$$
$$\texttt{GPR[Xlat(ry)]} \leftarrow \texttt{sign\_extend(temp}_{31..0}\texttt{)}$$

Exceptions:

None

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Set on Less Than **SLT**

15 – 11	10 – 8	7 – 5	4 – 0
RR 11101	rx	ry	SLT 00010
5	3	3	5

Format: `SLT rx, ry` **MIPS16e**

Purpose:

To record the result of a less-than comparison.

Description: `T ← (GPR[rx] < GPR[ry])`

The contents of GPR *ry* are subtracted from the contents of GPR *rx*. Considering both quantities as signed integers, if the contents of GPR *rx* are less than the contents of GPR *ry*, the result is set to 1 (true); otherwise, the result is set to 0 (false). This result is placed into GPR 24.

Restrictions:

None

Operation:

```
if GPR[Xlat(rx)] < GPR[Xlat(ry)] then
   GPR[24] ← 0^(GPRLEN-1) || 1
else
   GPR[24] ← 0^GPRLEN
endif
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Set on Less Than Immediate **SLTI**

15 — 11	10 — 8	7 — 0
SLTI 01010	rx	immediate
5	3	8

Format: `SLTI rx, immediate` **MIPS16e**

Purpose:

To record the result of a less-than comparison with a constant.

Description: `T ← (GPR[rx] < immediate)`

The 8-bit *immediate* is zero-extended and subtracted from the contents of GPR *rx*. Considering both quantities as signed integers, if GPR *rx* is less than the zero-extended *immediate*, the result is set to 1 (true); otherwise, the result is set to 0 (false). The result is placed into GPR 24.

Restrictions:

None

Operation:

```
if GPR[Xlat(rx)] < zero_extend(immediate) then
   GPR[24] ← 0^(GPRLEN-1) || 1
else
   GPR[24] ← 0^GPRLEN
endif
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Set on Less Than Immediate (Extended) — SLTI

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	imm 10:5	imm 15:11	SLTI 01010	rx	0 000	imm 4:0
5	6	5	5	3	3	5

Format: `SLTI rx, immediate` **MIPS16e**

Purpose:

To record the result of a less-than comparison with a constant.

Description: `T ← (GPR[rx] < immediate)`

The 16-bit *immediate* is sign-extended and subtracted from the contents of GPR *rx*. Considering both quantities as signed integers, if GPR *rx* is less than the sign-extended *immediate*, the result is set to 1 (true); otherwise, the result is set to 0 (false). The result is placed into GPR 24.

Restrictions:

None

Operation:

```
if GPR[Xlat(rx)] < sign_extend(immediate) then
   GPR[24] ← 0^(GPRLEN-1) || 1
else
   GPR[24] ← 0^GPRLEN
endif
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Set on Less Than Immediate Unsigned — SLTIU

15 — 11	10 — 8	7 — 0
SLTIU 01011	rx	immediate
5	3	8

Format: `SLTIU rx, immediate` **MIPS16e**

Purpose:

To record the result of an unsigned less-than comparison with a constant.

Description: `T ← (GPR[rx] < immediate)`

The 8-bit *immediate* is zero-extended and subtracted from the contents of GPR *rx*. Considering both quantities as unsigned integers, if GPR *rx* is less than the zero-extended *immediate*, the result is set to 1 (true); otherwise, the result is set to 0 (false). The result is placed into GPR 24.

Restrictions:

None

Operation:

```
if (0 || GPR[Xlat(rx)]) < (0 || zero_extend(immediate)) then
    GPR[24] ← 0^(GPRLEN-1) || 1
else
    GPR[24] ← 0^GPRLEN
endif
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Set on Less Than Immediate Unsigned (Extended) — SLTIU

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	imm 10:5	imm 15:11	SLTIU 01011	rx	0 000	imm 4:0
5	6	5	5	3	3	5

Format: `SLTIU rx, immediate` **MIPS16e**

Purpose:

To record the result of an unsigned less-than comparison with a constant.

Description: `T ← (GPR[rx] < immediate)`

The 16-bit *immediate* is sign-extended and subtracted from the contents of GPR *rx*. Considering both quantities as unsigned integers, if GPR *rx* is less than the sign-extended *immediate*, the result is set to 1 (true); otherwise, the result is set to 0 (false). The result is placed into GPR 24.

Restrictions:

None

Operation:

```
if (0 || GPR[Xlat(rx)]) < (0 || sign_extend(immediate)) then
    GPR[24] ← 0^(GPRLEN-1) || 1
else
    GPR[24] ← 0^GPRLEN
endif
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Set on Less Than Unsigned **SLTU**

15–11	10–8	7–5	4–0
RR 11101	rx	ry	SLTU 00011
5	3	3	5

Format: `SLTU rx, ry` **MIPS16e**

Purpose:

To record the result of an unsigned less-than comparison.

Description: `T ← (GPR[rx] < GPR[ry])`

The contents of GPR *ry* are subtracted from the contents of GPR *rx*. Considering both quantities as unsigned integers, if the contents of GPR *rx* are less than the contents of GPR *ry*, set the result to 1 (true); otherwise, set the result to 0 (false). The result is placed into GPR 24.

Restrictions:

None

Operation:

```
if (0 || GPR[Xlat(rx)]) < (0 || GPR[Xlat(ry)]) then
   GPR[24] ← 0^(GPRLEN-1) || 1
else
   GPR[24] ← 0^GPRLEN
endif
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Shift Word Right Arithmetic — SRA

15–11	10–8	7–5	4–2	1–0
SHIFT 00110	rx	ry	sa	SRA 11
5	3	3	3	2

Format: `SRA rx, ry, sa` **MIPS16e**

Purpose:

To execute an arithmetic right-shift of a word by a fixed number of bits—1 to 8 bits.

Description: `GPR[rx] ← GPR[ry] >> sa (arithmetic)`

The 32-bit contents of GPR *ry* are shifted right, and the sign bit is replicated into the emptied high-order bits. The 3-bit *sa* field specifies the shift amount. A shift amount of 0 is interpreted as a shift amount of 8. The result is sign-extended and placed into GPR *rx*.

Restrictions:

On 64-bit processors, if GPR *ry* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if (NotWordValue(GPR[Xlat(ry)])) then
    UNPREDICTABLE
endif
s ← 0^2 || sa
if (s = 0) then
    s ← 8
endif
temp ← (GPR[Xlat(ry)]_31)^s || GPR[Xlat(ry)]_31..s
GPR[Xlat(rx)] ← sign_extend(temp_31..0)
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Shift Word Right Arithmetic (Extended) SRA

31–27	26–22	21–16	15–11	10–8	7–5	4–2	1–0
EXTEND 11110	sa4:0	0 000000	SHIFT 00110	rx	ry	0 000	SRA 11
5	5	6	5	3	3	3	2

Format: `SRA rx, ry, sa` **MIPS16e**

Purpose:

To execute an arithmetic right-shift of a word by a fixed number of bits—0 to 31bits.

Description: `GPR[rx] ← GPR[ry] >> sa (arithmetic)`

The 32-bit contents of GPR *ry* are shifted right, and the sign bit is replicated into the emptied high-order bits. The 5-bit *sa* field specifies the shift amount. The result is sign-extended and placed into GPR *rx*.

Restrictions:

On 64-bit processors, if GPR *ry* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if (NotWordValue(GPR[Xlat(ry)])) then
    UNPREDICTABLE
endif
s ← sa
temp ← (GPR[Xlat(ry)]31)^s || GPR[Xlat(ry)]31..s
GPR[Xlat(rx)] ← sign_extend(temp31..0)
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Shift Word Right Arithmetic Variable **SRAV**

15–11	10–8	7–5	4–0
RR 11101	rx	ry	SRAV 00111
5	3	3	5

Format: `SRAV ry, rx` **MIPS16e**

Purpose:

To execute an arithmetic right-shift of a word by a variable number of bits.

Description: `GPR[ry] ← GPR[ry] >> GPR[rx] (arithmetic)`

The 32-bit contents of GPR *ry* are shifted right, and the sign bit is replicated into the emptied high-order bits; the word result is sign-extended and placed back in GPR *ry*. The 5 low-order bits of GPR *rx* specify the shift amount.

Restrictions:

On 64-bit processors, if GPR *ry* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if NotWordValue(GPR[ry]) then
    UNPREDICTABLE
endif
s ← GPR[Xlat(rx)]4..0
temp ← (GPR[Xlat(ry)]31)^s || GPR[Xlat(ry)]31..s
GPR[Xlat(ry)] ← sign_extend(temp31..0)
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

15–11	10–8	7–5	4–2	1–0
SHIFT 00110	rx	ry	sa	SRL 10
5	3	3	3	2

Format: `SRL rx, ry, sa` **MIPS16e**

Purpose:

To execute a logical right-shift of a word by a fixed number of bits—1 to 8 bits.

Description: `GPR[rx] ← GPR[ry] >> sa (logical)`

The 32-bit contents of GPR *ry* are shifted right, and zeros are inserted into the emptied high-order bits. The 3-bit *sa* field specifies the shift amount. A shift amount of 0 is interpreted as a shift amount of 8. The result is sign-extended and placed into GPR *rx*.

Restrictions:

On 64-bit processors, if GPR *ry* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if NotWordValue(GPR[ry]) then
    UNPREDICTABLE
endif
if sa = 0^3 then
    s ← 8
else
    s ← 0^2 || sa
endif
temp ← 0^s || GPR[Xlat(ry)]_31..s
GPR[Xlat(rx)] ← sign_extend(temp_31..0)
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Shift Word Right Logical (Extended) **SRL**

31–27	26–22	21–16	15–11	10–8	7–5	4–2	1–0
EXTEND 11110	sa4:0	0 000000	SHIFT 00110	rx	ry	0 000	SRL 10
5	5	6	5	3	3	3	2

Format: `SRL rx, ry, sa` **MIPS16e**

Purpose:

To execute a logical right-shift of a word by a fixed number of bits—0 to 31 bits.

Description: `GPR[rx] ← GPR[ry] >> sa (logical)`

The 32-bit contents of GPR *ry* are shifted right, and zeros are inserted into the emptied high-order bits. The 5-bit *sa* field specifies the shift amount. The result is sign-extended and placed into GPR *rx*.

Restrictions:

On 64-bit processors, if GPR *ry* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if NotWordValue(GPR[ry]) then
    UNPREDICTABLE
endif
s ← sa
temp ←  0^s || GPR[Xlat(ry)]_31..s
GPR[Xlat(rx)] ← sign_extend(temp_31..0)
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Shift Word Right Logical Variable **SRLV**

15 — 11	10 — 8	7 — 5	4 — 0
RR 11101	rx	ry	SRLV 00110
5	3	3	5

Format: `SRLV ry, rx` **MIPS16e**

Purpose:

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

Description: `GPR[ry] ← GPR[ry] >> GPR[rx] (logical)`

The 32-bit contents of GPR *ry* are shifted right, and zeros are inserted into the emptied high-order bits; the word result is sign-extended and placed back in GPR *ry*. The 5 low-order bits of GPR *rx* specify the shift amount.

Restrictions:

On 64-bit processors, if GPR *ry* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if (NotWordValue(GPR[Xlat(ry)])) then
    UNPREDICTABLE
endif
s ← GPR[Xlat(rx)]4..0
temp ← 0^s || GPR[Xlat(ry)]31..s
GPR[Xlat(ry)] ← sign_extend(temp31..0)
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Subtract Unsigned Word SUBU

15 — 11	10 — 8	7 — 5	4 — 2	1 — 0
RRR 11100	rx	ry	rz	SUBU 11
5	3	3	3	2

Format: `SUBU rz, rx, ry` **MIPS16e**

Purpose:

To subtract 32-bit integers.

Description: `GPR[rz] ← GPR[rx] - GPR[ry]`

The 32-bit word value in GPR *ry* is subtracted from the 32-bit value in GPR *rx* and the 32-bit arithmetic result is sign-extended and placed into GPR *rz*.

No integer overflow exception occurs under any circumstances.

Restrictions:

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

Operation:

```
if(NotWordValue(GPR[Xlat(rx)])or NotWordValue(GPR[Xlat(ry)]))then
    UNPREDICTABLE
endif
temp ← GPR[Xlat(rx)] - GPR[Xlat(ry)]
GPR[Xlat(rz)] ← sign_extend(temp31..0)
```

Exceptions:

None

Programming Notes:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Word **SW**

15 … 11	10 … 8	7 … 5	4 … 0
SW 11011	rx	ry	offset
5	3	3	5

Format: `SW ry, offset(rx)` **MIPS16e**

Purpose:

To store a word to memory.

Description: `memory[GPR[rx] + offset] ← GPR[ry]`

The 5-bit *offset* is shifted left 2 bits, zero-extended, and then added to the contents of GPR *rx* to form the effective address. The contents of GPR *ry* are stored at the effective address.

Restrictions:

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

Operation:

```
vAddr  ← zero_extend(offset || 0^2) + GPR[Xlat(rx)]
if vAddr_1..0 ≠ 0^2 then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor (ReverseEndian || 0^2))
bytesel← vAddr_2..0 xor (BigEndianCPU || 0^2)
datadoubleword← GPR[Xlat(ry)]_63-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Word (Extended) **SW**

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	SW 11011	rx	ry	offset 4:0
5	6	5	5	3	3	5

Format: `SW ry, offset(rx)` **MIPS16e**

Purpose:

To store a word to memory.

Description: `memory[GPR[rx] + offset] ← GPR[ry]`

The 16-bit *offset* is sign-extended and then added to the contents of GPR *rx* to form the effective address. The contents of GPR *ry* are stored at the effective address.

Restrictions:

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

Operation:

```
vAddr  ← sign_extend(offset) + GPR[Xlat(rx)]
if vAddr_1..0 ≠ 0^2 then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor (ReverseEndian || 0^2))
bytesel← vAddr_2..0 xor (BigEndianCPU || 0^2)
datadoubleword← GPR[Xlat(ry)]_63-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Word rx (SP-Relative) **SW**

15 … 11	10 … 8	7 … 5 4 … 0
SWSP 11010	rx	offset
5	3	8

Format: `SW rx, offset(sp)` **MIPS16e**

Purpose:

To store an SP-relative word to memory.

Description: `memory[GPR[sp] + offset] ← GPR[rx]`

The 8-bit *offset* is shifted left 2 bits, zero-extended, and then added to the contents of GPR 29 to form the effective address. The contents of GPR *rx* are stored at the effective address.

Restrictions:

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

Operation:

```
vAddr  ← zero_extend(offset || 0^2) + GPR[29]
if vAddr_1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor (ReverseEndian || 0^2))
bytesel← vAddr_2..0 xor (BigEndianCPU || 0^2)
datadoubleword← GPR[Xlat(rx)]_63-8*bytesel..0 || 0^(8*bytesel)
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Word rx (SP-Relative, Extended) SW

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	SWSP 11010	rx	0 000	offset 4:0
5	6	5	5	3	3	5

Format: `SW rx, offset(sp)` **MIPS16e**

Purpose:

To store an SP-relative word to memory.

Description: `memory[GPR[sp] + offset] ← GPR[rx]`

The 16-bit *offset* is sign-extended and then added to the contents of GPR 29 to form the effective address. The contents of GPR *rx* are stored at the effective address.

Restrictions:

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

Operation:

```
vAddr ← sign_extend(offset) + GPR[29]
if vAddr1..0 ≠ 0^2 then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
bytesel← vAddr2..0 xor (BigEndianCPU || 0^2)
datadoubleword← GPR[Xlat(rx)]63-8*bytesel..0 || 0^(8*bytesel)
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Word ra (SP-Relative) — SW

15–11	10–8	7–0
I8 01100	SWRASP 010	offset
5	3	8

Format: `SW ra, offset(sp)` **MIPS16e**

Purpose:

To store register *ra* SP-relative to memory.

Description: `memory[sp + offset] ← ra`

The 8-bit *offset* is shifted left 2 bits, zero-extended, and then added to the contents of GPR 29 to form the effective address. The contents of GPR 31 are stored at the effective address.

Restrictions:

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

Operation:

```
vAddr  ← zero_extend(offset || 0^2) + GPR[29]
if vAddr_1..0 ≠ 0^2 then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..3 || (pAddr_2..0 xor (ReverseEndian || 0^2))
bytesel← vAddr_2..0 xor (BigEndianCPU || 0^2)
datadoubleword← GPR[31]_63-8*bytesel..0 || 0^(8*bytesel)
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Store Word ra(SP-Relative, Extended) SW

31–27	26–21	20–16	15–11	10–8	7–5	4–0
EXTEND 11110	offset 10:5	offset 15:11	I8 01100	SWRASP 010	0 000	offset 4:0
5	6	5	5	3	3	5

Format: `SW ra, offset(sp)` **MIPS16e**

Purpose:

To store register *ra* SP-relative to memory.

Description: `memory[sp + offset] ← ra`

The 16-bit *offset* is sign-extended and then added to the contents of GPR 29 to form the effective address. The contents of GPR 31 are stored at the effective address.

Restrictions:

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

Operation:

```
vAddr  ← sign_extend(offset) + GPR[29]
if vAddr1..0 ≠ 0^2 then
    SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2))
bytesel← vAddr2..0 xor (BigEndianCPU || 0^2)
datadoubleword← GPR[31]63-8*bytesel..0 || 0^(8*bytesel)
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Exclusive OR **XOR**

15 – 11	10 – 8	7 – 5	4 – 0
RR 11101	rx	ry	XOR 01110
5	3	3	5

Format: `XOR rx, ry` **MIPS16e**

Purpose:

To do a bitwise logical Exclusive OR.

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

The contents of GPR *ry* are combined with the contents of GPR *rx* in a bitwise Exclusive OR operation. The result is placed in GPR *rx*.

Restrictions:

None

Operation:

```
GPR[Xlat(rx)] ← GPR[Xlat(rx)] xor GPR[Xlat(ry)]
```

Exceptions:

None

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Zero-Extend Byte **ZEB**

15 … 11	10 … 8	7 … 5	4 … 0
RR 11101	rx	ZEB 000	CNVT 10001
5	3	3	5

Format: `ZEB rx` **MIPS16e**

Purpose:

Zero-extend least significant byte in register rx.

Description: `GPR[rx] ← zero_extend(GPR[rx]`$_{7..0}$`);`

The least significant byte of rx is zero-extended and the value written back to rx.

Restrictions:

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

Operation:

```
if NotWordValue(GPR[Xlat(rx)]) then
    UNPREDICTABLE
endif
temp ← GPR[Xlat(rx)]
GPR[Xlat(rx)] ← 0 || temp7..0
```

Exceptions:

None

Programming Notes:

None.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Zero-Extend Halfword **ZEH**

15 ... 11	10 ... 8	7 ... 5	4 ... 0
RR 11101	rx	ZEH 001	CNVT 10001
5	3	3	5

Format: `ZEH rx` **MIPS16e**

Purpose:

Zero-extend least significant halfword in register rx.

Description: `GPR[rx] ← zero_extend(GPR[rx]15..0);`

The least significant halfword of rx is zero-extended and the value written back to rx.

Restrictions:

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

Operation:

```
if NotWordValue(GPR[Xlat(rx)]) then
   UNPREDICTABLE
endif
temp ← GPR[Xlat(rx)]
GPR[Xlat(rx)] ← 0 || temp15..0
```

Exceptions:

None

Programming Notes:

None.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Zero-Extend Word **ZEW**

15 … 11	10 … 8	7 … 5	4 … 0
RR 11101	rx	ZEW 010	CNVT 10001
5	3	3	5

Format: `ZEW rx` **MIPS16e (64-bit only)**

Purpose:

Zero-extend least significant word in register rx.

Description: `GPR[rx] ← Zero_extend(GPR[rx]`$_{31..0}$`);`

The least significant word of rx is Zero-extended and the value written back to rx.

Restrictions:

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

Operation:

```
if NotWordValue(GPR[Xlat(rx)]) then
    UNPREDICTABLE
endif
temp ← GPR[Xlat(rx)]
GPR[Xlat(rx)] ← 0^32 || temp31..0
```

Exceptions:

Reserved Instruction

Programming Notes:

None.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Revision History

In the left hand page margins of this document you may find vertical change bars to note the location of significant changes to this document since its last release. Significant changes are defined as those which you should take note of as you use the MIPS IP. Changes to correct grammar, spelling errors or similar may or may not be noted with change bars. Change bars will be removed for changes which are more than one revision old.

Please note: Limitations on the authoring tools make it difficult to place change bars on changes to figures. Change bars on figure titles are used to denote a potential change in the figure itself.

Revision	Date	Description
0.90	November 1, 2000	External review copy of reorganized and updated architecture documentation.
0.91	November 15, 2000	Changes in this revision: • Correct table 3-10 description of branch instructions (branches really are implemented in the 32-bit architecture and are extensible) • Correct the pseudo code for all MIPS16 branches - the offset value thould be added to the address of the instruction following the branch, not the branch itself.
0.92	December 15, 2000	Changes in this revision: • Add missing I8_MOVER32 instruction format.
0.93	January 25, 2001	Changes in this revision: • Correct minor typos in the previous version. • Add the 32-bit MIPS version of JALX and update the instruction descriptions of JAL and JALX.
0.95	March 12, 2001	Document cleanup for next external release.
0.96	November 12, 2001	Changes in this revision: • Declassify the MIPS32 Architecture for Programmers volume. • Fix PDF bookmarks for the MIPS16 instructions. • Fix formatting in instruction translation section. • Correct the description of the shift count for extended SRA and SLL. • Change all uses of “MIPS16” to “MIPS16e”.
1.00	August 29, 2002	Changes in this revision: • Update pseudo code for SAVE and RESTORE to be explicit about the memory operations inherent in the instructions. • Correct extended PC-relative LW and LD to remove the implication that they can be executed in the delay slot of a jump. • Add section defining instruction fetch restrictions when the processor is running in MIPS16e mode and the fetch address is in uncached memory.

Copyright © 2005 MIPS Technologies Inc. All rights reserved.

Revision	Date	Description
2.00	May 15, 2003	Changes in this revision: • For MIPS64 processors, add a programming note to ADDIUPC to indicate that this instruction will generate the expected result only when run in the 32-bit Compatibility Address Space. • For MIPS64 processors, clean up the input operand sign-extension requirements for ADDIUPC, ADDIUSP, ADDU, NEG, SEB, SEH, SEW, ZEB, ZEH, and ZEW. • Add a note to specify that the ISA Mode flag is made available to software in *EPC*, *ErrorEPC*, or *DEPC* when an exception occurs. • Clarify that for the purposes of Watchpoints and EJTAG Breakpoints, that PC-releative load references are consider data, not instruction, references.
2.50	July 1, 2005	Changes in this revision: • Make it explicit that attempting to execute a non-extensible instruction must cause a Reserved Instruction exception. This was implied, but not explicitly stated in the previous revision of the document. • Update all files to FrameMaker 7.1. •

Copyright © 2005 MIPS Technologies Inc. All rights reserved.