TOSHIBA

32Bit TX System RISC
TX19A Family
Architecture

Rev １.0

TOSHIBA CORPORATION

Semiconductor Company

- The information contained herein is subject to change without notice. 021023_D

- TOSHIBA is continually working to improve the quality and reliability of its products. Nevertheless, semiconductor devices in general can malfunction or fail due to their inherent electrical sensitivity and vulnerability to physical stress. It is the responsibility of the buyer, when utilizing TOSHIBA products, to comply with the standards of safety in making a safe design for the entire system, and to avoid situations in which a malfunction or failure of such TOSHIBA products could cause loss of human life, bodily injury or damage to property.
  In developing your designs, please ensure that TOSHIBA products are used within specified operating ranges as set forth in the most recent TOSHIBA products specifications. Also, please keep in mind the precautions and conditions set forth in the "Handling Guide for Semiconductor Devices," or "TOSHIBA Semiconductor Reliability Handbook" etc. 021023_A

- The TOSHIBA products listed in this document are intended for usage in general electronics applications (computer, personal equipment, office equipment, measuring equipment, industrial robotics, domestic appliances, etc.). These TOSHIBA products are neither intended nor warranted for usage in equipment that requires extraordinarily high quality and/or reliability or a malfunction or failure of which may cause loss of human life or bodily injury ("Unintended Usage"). Unintended Usage include atomic energy control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, combustion control instruments, medical instruments, all types of safety devices, etc. Unintended Usage of TOSHIBA products listed in this document shall be made at the customer's own risk. 021023_B

- The products described in this document shall not be used or embedded to any downstream products of which manufacture, use and/or sale are prohibited under any applicable laws and regulations. 060106_Q

- The information contained herein is presented only as a guide for the applications of our products. No responsibility is assumed by TOSHIBA for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of TOSHIBA or the third parties. 021023_C

- The products described in this document are subject to foreign exchange and foreign trade control laws. 060925_E

- MIPS16,Application Specific Extensions & R3000A are trademarks of MIPS Group , a division of Silicon Graphics, Inc.

Preface

This manual describes the architecture of the Toshiba TX19A family.

Contents

Chapter 1: Introduction
Outline of TX19A
Chapter 2: CPU Architecture Overview
-Data load in the CPU registers and memory
-Overview of the functionality of the registers
Chapter 3: 32-Bit ISA Summary and Programming Tips
-Summary of the 32-bit instruction set architecture (ISA)
Chapter 4: 16-Bit ISA Summary and Programming Tips
-Summary of the 16-bit ISA
Chapter 5: CPU Pipeline
-Information about the instruction pipeline
Chapter 6: Memory Management
-The virtual and physical address spaces and these mapping manners
Chapter 7: Internal I/O Bus Operation
-Outlines of the Harvard architecture and the protocols for internal bus transactions
Chapter 8: System Control Coprocessor (CP0) Registers
-A group of registers associated with system configuration and exception processing
Chapter 9: CPU Exception Processing
-The events that cause exceptions and the sequences to be handled
Chapter 10: Power Consumption Management
-The methods of dynamically controlling power consumption during operation
Appendix A: 32-Bit ISA Details
-Detailed description of each instruction available in 32-bit ISA mode
Appendix B: 16-Bit ISA Details
-Detailed description of each instruction available in 16-bit ISA mode
Appendix C: Programming Restrictions
-The restrictions need to be observed in writing assembly-language programs
Appendix D: Compatibility Among TX19, TX19A and TX39 Architectures
-Provides comparisons among the three RISC processor families
Appendix E: 32-Bit ISA Instruction Bit Encoding
-The opcode bit encoding for the 32-bit ISA
Appendix F: 16-Bit ISA Instruction Bit Encoding
-the opcode bit encoding for the 16-bit ISA

March.2007

Readers

This manual is written for software and hardware developers who want to develop products using TX19A processors and controllers.
RISC processors including TX19A have a number of features that make them stand out from CISC processors. If you are unfamiliar with RISC architecture, Chapter 1 should be useful for you. Please note that RISC processors have a small instruction set. There are no complex instructions such as LDIR (block transfer), CPIR (block search), BS1B (bit scan). Since RISC has very few instructions, a programmer or a compiler needs to implement additional instructions by using available RISC instructions.
Chapter 2, the architecture overview, should help programmers who can use a high-level language such as C in developing software.
Assembly language programmers must be well versed in the intricacies of the machine architecture. The performance of software systems is drastically affected by how well software designers understand the basic hardware technologies at work in a system. Therefore, we recommend assembly language programmers to read the entire manual that gives a detailed description of the TX19A architecture for overall understanding.

Related Document

Semiconductor Reliability Handbook (Integrated Circuits)
This book describes the methodology used by Toshiba to achieve robust semiconductor designs before market introduction and to ensure high quality and reliability in volume production phase.

Chapter 1 Introduction

This chapter provides the features of the TX19A and a general description of how the TX19A RISC design differs from CISC processors such as the Toshiba 900/L1.

1.1 Processor General Features

The TX19A, a high quality 32-bit RISC processor, is created based on MIPS Technologies Inc.'s R3000A architecture that contains reduced code size of 16 bit architecture "MIPS16e-TX". The instruction set of the TX19A includes the 32-bit instructions of the TX39 as a subset. Thus the TX19A software preserves upward compatibility with TX39 and TX19.
The TX19A family of integrated processors and controllers is built on the TX19A core processor, an on-chip bus and a selection of intelligent peripherals appropriate for specific applications. The TX19A is available as an ASIC-ready core and a family of standard ASSP products.

Instruction sets of MIPS 16e-TX and MIPS S32

- MIPS16e-TX instruction sets are object-code compatible with MIPS16 ASE except for the area that Toshiba extended MIPS16 ASE with permission of MIPS Technologies, Inc.
  Note: The TX19A does not provide support for MIPS16 ASE instructions for 64-bit operations.
- The 32-bit instructions are object-code compatible with the high-performance TX39 family.
  -Switchable run-time between 16-bit and 32-bit ISA modes through an instruction. These conditions are respectively called as 16 bit ISA mode and 32 bit ISA mode.
  -Hardware interlocks enables to send an instruction to refer the data loaded in register immediately after the load instruction. This eliminates the need to insert a NOP (No Operation) instruction.
  -Branch-likely instructions allow the processor to execute the instruction at the target location immediately after the branch instruction. This eliminates the need to insert a NOP instruction.

High Performance

- Single clock cycle execution for most instructions
- 3-operand computational instructions
- Full 32-bit operations: Contains 32-bit general-purpose registers and a 32-bit program counter.
- 8 sets of 32 general-purpose registers (shadow register sets): Automatically switched on entry to an interrupt, based on its priority level.
- 5-stage pipeline
- Independent on-chip instruction and data memory with an access time of one clock cycle applicable
- An on-chip write buffer applicable
- Harvard architecture

The TX19A uses separate buses for code and data operands. In the TX19A, there are four sets of buses: a data bus for carrying data (operands) in and out of the processor core, an address bus for

accessing data operands, a bus to carry the opcodes and an address bus to access the opcodes. The ability to access code and data simultaneously through separate buses increases instruction throughput.

- Nonblocking loads function enables to execute the subsequent instruction in a load delay slot in case a large latency appeared during data loading from external memory.
- On-chip multiplier/accumulator (MAC): Executes a (32-bit x 32-bit + 64-bit) and (64-bit – 32-bit x 32-bit) operations in a single clock cycle.
- 4-Gbyte virtual address space
- Integrated coprocessor: The TX19A contains the system control coprocessor (CP0) for system configuration, exception handling and memory management.

Low Power

- Power-optimized design

Programmable power management modes (Halt and Doze): In Doze mode, the processor senses external bus requests.

Real-Time Interrupt Response

- Distinct starting locations for each interrupt service routine
- Automatically generated vectors for each interrupt source: Interrupt priorities are resolved upon reading the exception vector. Interruption exception is executed when its priority level is higher than the current one. It makes the TX19A effective for quick response to an interrupt request that needs immediate action.
- On an interrupt, register sets are automatically switched based on its priority level.

Processor Core for System ASIC Applications

- Unified manufacturing process and development environment as ASIC
- Compact core design
- The processor core can be directly connected to the G-Bus, the standard on-chip bus for the TX series.

System Development Environments

- Language tools: C compilers and assemblers
  Both Toshiba's proprietary and third-party tools are offered.
- Real-time operating systems
  Both Toshiba's proprietary (⌈ITRON) and third-party real-time operating systems are offered.
- Debug support systems
  -Both Toshiba's proprietary and third-party real-time emulators are offered to support source-level debugging.

-Support for utility software to insert debug support unit (DSU) circuitry into an ASIC design.

1.2 What Is RISC?

Until the early 1980s, all CPUs followed the complex instruction set computer (CISC) design philosophy. To preserve compatibility with the existing pool of software, CISC processors evolved by adding new types of machine instructions and more intricate operations. Generally, CISC refers to CPUs with hundreds of instructions designed for every possible situation. Designing CPUs with hundreds of instructions not only requires many transistors but is also very complicated, timing consuming and expensive.

In the early 1980s, a controversy broke out in the computer design community. Proponents of a new type of computer design argued that no one was using so many instructions. As it was developed, it came to be known as reduced instruction set computer (RISC). RISC concepts emerged by statistical analysis of how software actually uses the resources of a processor. According to experiments, many of the complex instructions were never used by programmers and compilers. The huge costs of implementing numerous instructions made some designers think of streamlining the instruction set.

■ Feature 1 Simple instructions

RISC processors have a small instruction set. For example, there are no such complex instructions as block transfer, block search, bit scan and so forth.

Additionally, RISC uses the load/store architecture. In CISC processors, data can be manipulated while it is still in memory. For example, “ADD A, (1000H)” contained in 16-bit CISC processor TLCS-900/L1, is an instruction to bring the contents memory location 1000H into the CPU, sum it up with data in register A and store the total in A. RISC did away with this kind of instructions. In RISC, a single instruction can either load from memory into a register or store from a register into memory. In other words, all operations are performed on operands held in CPU registers.

Since CISC processors have a large number of instructions, each with so many different addressing modes, microcode is used to implement all of them. This feature of CISC makes the job of programmers easy and helps to reduce code size. However, the implementation of microcode requires more space on chip, creating a bottleneck in an effort to improve processor performance.

■ Feature 2 Fixed instruction size

RISC processors have a fixed instruction size. In a CISC microprocessor, instructions can be 1, 2 or even 7 bytes at the maximum. This variable instruction size makes the task of the instruction decoder very complicated since the size of the incoming instruction can never be known. In the TX19A microprocessor, the instruction size is fixed at 32 bits. The fixed instruction size enables the CPU to decode instructions quickly.

■ **Feature 3 Heavily pipelined**

Since RISC has only a limited number of simple instructions, most of the instructions can be executed in one clock cycle. Therefore, RISC is easier to pipeline than CISC that requires a different number of clock cycles for each instruction in pipeline. Generally, RISC processors are heavily pipelined.

1.3 Features of the TX19A

The previous section provided an overview of the RISC features which are different from CISC processors. In this section, we explore how the instruction set architecture (ISA) is implemented in the TX19A in comparison to the 870/X and the 900/L1, 8-bit and 16-bit CISC processors from Toshiba.
The TX19A has two ISA modes, 16-bit and 32-bit. The condition that each mode is executed is respectively called as 16 bit ISA mode and 32 bit ISA mode. It provides for efficient run-time switching between 16-bit and 32-bit ISA modes through an instruction. The 16-bit instruction set (MIPS16e+) is not a separate instruction set indeed but a 16-bit extension of the full 32-bit MIPS architecture. The 32-bit ISA has 103 instructions, the 16-bit ISA 128 instructions. Programs will consist of procedures in 16-bit mode for density or in 32-bit mode for performance.
On the other hand, the 870/X and the 900/L1 are both CISC processors having nearly 1000 types of instructions and many addressing modes. CISC processors are, in general, excel in code efficiency.

1.3.1 Instruction Set Architecture

- **The TX19A did away with complex instructions.**

The TX19A has only the basic instructions such as load, store, add, subtract, multiply, divide, AND, OR, XOR, shift, jump and branch. There are no complex instructions like LDIR (block transfer) and CPIR (block search) available with the 900/L1. It is the responsibility of the compiler (or the programmer) to generate software routines to perform complex instructions that are done in hardware by CICS processors. As exceptions, are the multiply-and-add (MADD and MADDU) and multiply-and-subtract (MSUB and MSUBU) instructions that require very fast processing are included in instruction sets (these instructions are executed by the dedicated MAC circuitry.)

- **The TX19A did away with instructions that can be implemented by some other instructions**

To reduce the size of the instruction set, the TX19A aggressively eliminated the instructions that can be implemented using other instructions. For example, the TX19A does not have the NOP (No Operation), INC (Increment) and DEC (Decrement) instructions. Instead of NOP, a shift instruction can be used as shown below for TX19A processors:

```
SLL    r0,r0,0
```

In the TX19A, register r0 is hardwired to a constant value of 0. The above instruction actually shifts the contents of r0 by zero bits and places the result back in r0. (The assembler permits NOP as a

pseudoinstruction for program readability; however, it turns NOP into a shift instruction.)
A register increment can be implemented by using the ADDIU (Add Immediate Unsigned) instruction as shown below:

ADDIU *rt*,*rs*,1

In this condition, *rt* and *rs* are the target and source registers respectively. Likewise, a register decrement can be implemented as follows:

ADDIU *rt*,*rs*,-1

- **The TX19A discarded instructions synthesizable from two or more simple instructions**

The TX19A further pared down the instruction set by discarding the instructions that can be performed by two or more simple instructions. For example, the TX19A does not have the POP and the PUSH instructions for accessing the stack. In CISC processors, as a PUSH instruction is executed, the contents of a register is saved on the stack and the stack pointer register is decremented by the amount of the register size. In the TX19A, one of the 32 general-purpose registers is used as a stack pointer; pushing onto the stack is accomplished by executing an add instruction on the stack pointer and a store instruction.

- **The TX19A uses the load/store architecture**

In CISC processors such as the 870/X and the 900/L1, data can be manipulated while it is still in memory, like ADD A, (1000H). The TX19 did away with this kind of instructions; in the TX19, the load and store instructions are the only instructions that move data between memory and CPU general registers. However, the TX19A enhanced the capability of the TX19 by adding a group of instructions that manipulate a specific bit in memory or add an immediate to a value in memory.

- **The TX19A has only a few memory addressing modes**

The 900/L1 and the 870/X1 have seven or more addressing modes for memory accesses. For example, there are register indirect, register indirect with autoincrement, indexed relative, based indexed relative, etc. These versatile addressing modes are very useful for assembly language programmers and contribute to a reduction in code size.
In contrast, in 32-bit ISA mode, the TX19A has only one addressing mode for accessing memory locations in order to simplify hardware implementation: i.e., based relative. In 16-bit ISA mode, the TX19A has three more addressing modes called PC-relative, SP-relative and FP-relative; only three 16-bit instructions can use PC-and SP-relative addressing modes, however.

- **The TX19A has three-operand computational instructions 3**

In the TX19A, many computational instructions use triadic format. In triadic instruction format, there are two source registers and one destination register. An example of triadic format is:

ADD *rd*,*rs1*,*rs2*

This instruction adds the contents of two source registers, *rs1* and *rs2*, and stores the results in *rd*. On the other hand, the 900/L1 adds the contents of XWA and XBC and puts the result in XWA.

ADD XWA,XBC

- **The TX19A does not have a flag register**

The TX19A does not have a dedicated flag register with the carry, overflow and sign bits. For example, in the 900/L1, the carry flag is used to indicate whether or not there was a carry from an addition or a borrow as a result of subtraction. It is widely used in multibyte additions and subtractions. The 900/L1 has the ADC instruction to add the carry bit to the sum of two registers.
On the other hand, the TX19A can perform 32-bit additions at a time; so the flag bit is rarely needed. To perform an add-with-carry, a routine must first explicitly determine whether the addition has resulted in a carry, and then record the occurrence of a carry in a register. When doing multiword additions, two different code sequences are required: one for adding with a carry-in and one for adding without a carry-in.
Additionally, the 900/L1 CP (compare) instruction uses the carry flag to indicate whether or not there was a borrow as a result of subtraction. In the TX19A, the result of compare instructions such as SLT (Set On Less Than) is placed into a general register.

1.3.2 Instruction Format

The TX19A has two ISA modes, 16-bit and 32-bit. All the instructions for the 32-bit ISA mode, as the name suggests, consist of 32 bits. All the instructions for the 16-bit ISA mode consist of 16 bits, with a few exceptions. The 870/C instructions have the variable length: 1, 2, 3, 4, 5 and 6 bytes. Furthermore, the 900/L1 covers 7 byte-instruction as the longest. This variable instruction length is useful to reduce code size; however, it makes the task of the instruction decoder very complicated and slow.

1.3.3 Instruction Pipelines

The TX19A has a five-stage pipeline. The five-stage pipeline divides the execution of each instruction into five discrete portions and executes up to five instructions simultaneously. Each stage takes one clock cycle.
The major characteristics of the TX19A is that the execution of most instructions requires a uniform number of clock cycles; thus the TX19A is relatively easy to pipeline. The TX19A achieves an instruction execution rate approaching one instruction per clock cycle.

If the instruction stream includes a variety of different instruction lengths as in CISC processors, pipeline management becomes very complicated. Moreover, such a varied, complex instruction stream makes it almost impossible for a compiler to schedule instructions to reduce or eliminate pipeline stalls.

Chapter 2 CPU Architecture Overview

This chapter outlines the TX19A architecture, data formats, programming model, ISA modes, coprocessors, instruction pipeline and memory management.

2.1 Data Formats

This section describes the organization of data in registers and memory and how operands are signor zero-extended for operations.

2.1.1 Byte Ordering

The TX19A supports many data types including 8-bit, 16-bit, 32-bit and 64 bit. A byte is defined as 8 bits. A halfword is two bytes, or 16 bits. A word is four bytes, or 32 bits. A doubleword is two words, or 64 bits.

For multibyte data types, the TX19A supports both big-endian and little-endian formats. Byte ordering (endianness) can be set through the ENDIAN input pin during a reset sequence. (In some TX19A components, byte ordering is fixed to either big-endian or little-endian.)

Figure 2-1 shows the ordering of bytes in a word for big-endian and little-endian formats. The TX19A processor uses a byte addressing. The big-endian ordering assigns the lowest address to the highest-order (leftmost) byte. The little-endian ordering assigns the lowest address to the lowest-order (rightmost) byte. Notice that, in the little-endian format, each byte of a multibyte integer is placed in the same memory location regardless of whether the integer is defined as a halfword or a word in size.

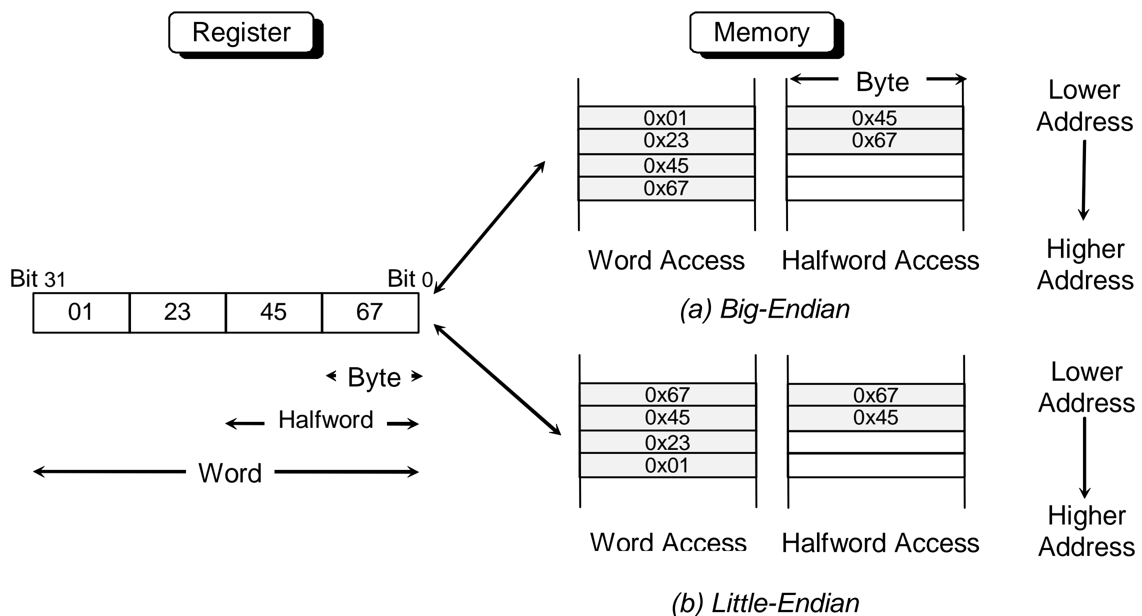


Figure 2-1 Byte Ordering

2.1.2 Aligned and Misaligned Accesses

The TX19A uses byte addressing for byte, halfword and word accesses. The address of a multibyte data item is the address of the lowest memory location for that data item; i.e. the address of the most-significant byte on a big-endian configuration and the address of the least-significant byte on a little-endian configuration.

Memory access instructions have a natural alignment boundary equal to the operand length (see fig. 2-2). In other words, the natural address of an operand is an integer multiple of the operand length. A memory operand is aligned if its address is a multiple of two for halfword accesses or a multiple of four for word accesses.

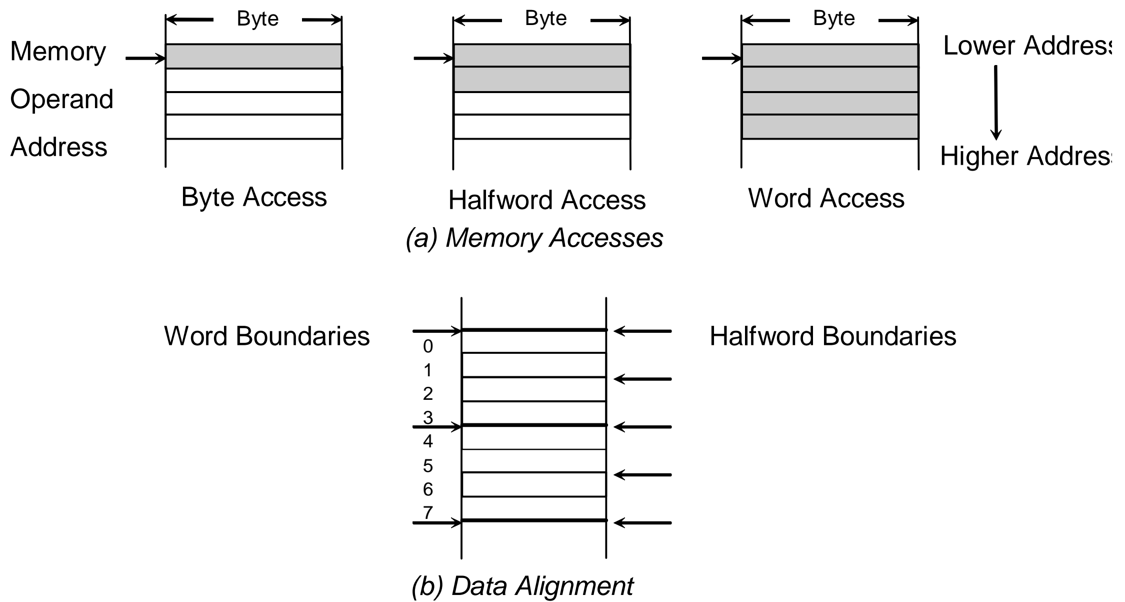
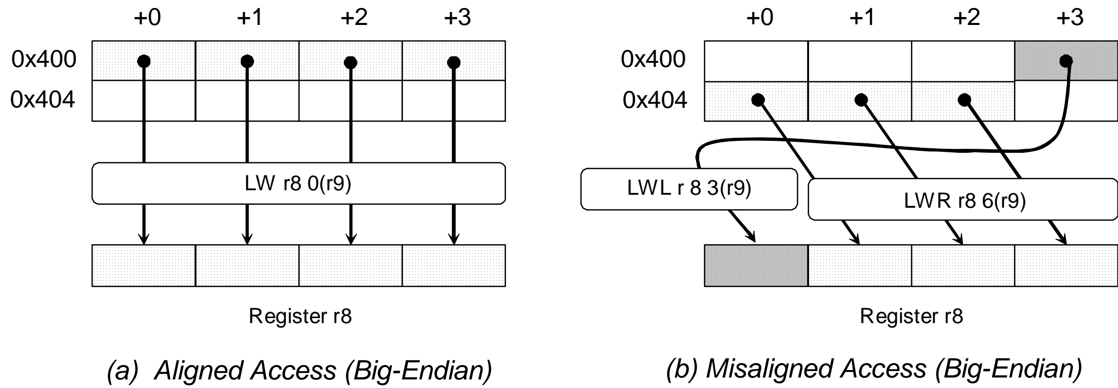


Figure 2-2 Aligned Data Items

Most instructions require their memory operands to be aligned because alignment affects performance. Special instructions are provided for addressing words that cross a boundary between two words: LWL (Load Word Left), LWR (Load Word Right), SWL (Store Word Left) and SWR (Store Word Right). These instructions are used in pairs. Figure 2-3 illustrates how a word of aligned and misaligned data is loaded from memory into a CPU register.



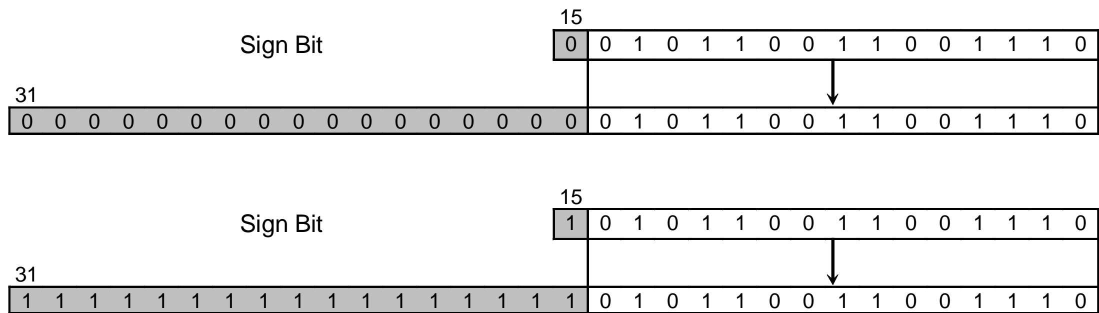
(a) Aligned Access (Big-Endian)

(b) Misaligned Access (Big-Endian)

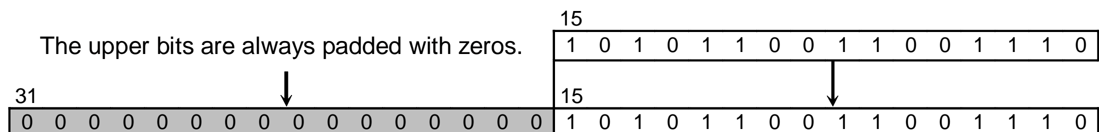
Figure 2-3 Aligned and Misaligned Accesses

2.1.3 Data Extensions

Figure 2-4 illustrates sign extension and zero extension. In signed numbers, the most-significant bit is the sign and the remaining bits are set aside for the magnitude of the number. Sign extension copies the most-significant bit (i.e., sign bit) of a 16-bit immediate or the loaded byte or halfword into the upper bits. Zero extension fills unused bits in a word with zeros irrespective of the value of the most-significant bit of a 16-bit immediate or the loaded byte or halfword.



(a) 16-Bit to 32-Bit Sign Extension



(b) 16-Bit to 32-Bit Zero Extension

Figure 2-4 Sign Extension and Zero Extension

Sign extension is typically used to avoid problems associated with arithmetic operations. For example, the ADDI (Add Immediate Signed) instruction only can take a 16-bit immediate. The instruction "ADDI r3, r1, 0x1234" sign extends 0x1234 and adds it to the contents of register r1 to form a 32-bit result. The result is placed into register r3.
The TX19A also applies sign extension to such instructions as LB (Load Byte), LBU (Load Byte Unsigned) LH (Load Halfword), LHU (Load Halfword Unsigned) LW (Load Word), SB (Store Byte), SH (Store Halfword), SW (Store Word) since the only addressing mode supported is base register plus 16-bit immediate (i.e., offset). For example, the instruction "LB r9, 4(r8)" sign-extends the offset (4 or binary 0100) and adds it to the contents of the base address held in r8 to form an effective address. The word in the addressed memory location is loaded into r9.
To load byte data and halfword data in register, sign extension or zero extension is selected depend on the instructions. Therefore, the LB and LH instructions sign- extend the loaded byte and put it in the target register; the LBU instruction zero-extends the loaded byte.
Additionally, there are two types of logical AND and logical OR instructions each, AND/ANDI and OR/ORI. The AND and OR instructions perform AND and OR operations with word data whereas the ANDI (AND Immediate) and ORI (OR Immediate) perform AND and OR operations with word data and halfword data. ANDI and ORI zero-extends a 16-bit immediate and combine it with the contents of a general register in a bitwise logical AND or OR operation.

2.2 Programming Model

The TX19A programming model consists of two groups of registers: CPU registers and system control coprocessor (CP0) registers.

2.2.1 CPU Registers

Figure 2-5 shows the CPU registers. The TX19A has eight sets of 32 general-purpose registers (GPRs) called shadow sets for a total of 256 GPRs, a program counter (PC) register and two special registers (HI/LO) that hold the results of integer multiply and divide operations. All CPU registers are 32 bits in length.

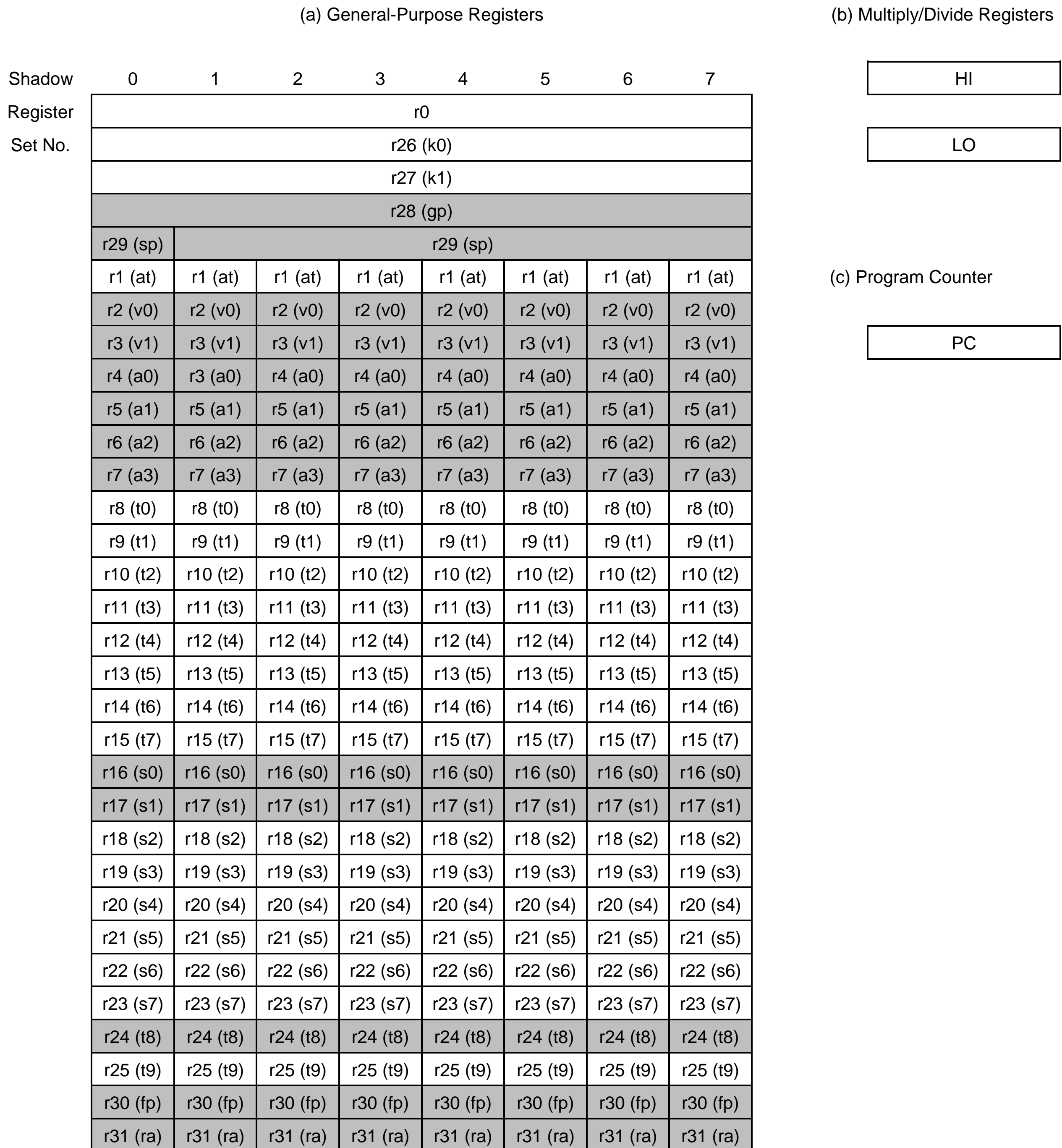


Figure 2-5 CPU Registers

■ General-Purpose Registers

The TX19A core processor contains eight sets of general-purpose registers known as the "Shadow Register Sets" numbered 0 to 7. Each shadow set consists of 32 registers (r0 to r31), except that all the shadow sets have r0, r26, r27 and r28 in common and that shadow sets 1 to 7 have r29 in common. All the other general-purpose registers are available in each shadow set. Switching to a new shadow set is automatically done by processor hardware via interrupt or can be done with an instruction (MTC0).

The 32-bit ISA instructions can use any of the general-purpose registers shown in Figure 2-5. The general registers are numbered from r0 to r31. The general registers except r0 have symbol names (software names) like v0-v1, a0-a3, and so on that are used by an assembler. The 32-bit ISA instructions treat the general registers symmetrically, with the exception of r0 and r31. r0 is hardwired to a value of 0. As such, r0 can be used by any instruction as a target register when the result of an operation is to be discarded or as a source register when a zero value is necessary. r31 (ra: return address) is a link register used by Jump-and-Link, Branch-and-Link and Branch-Likely and-Link instructions. These instructions are to store an address, which shows the restarting point after a subroutine has been executed, in r31.

In the 16-bit instructions, only eight of the 32 general-purpose registers are normally visible, r2 to r7, r16 and r17. Since the processor includes the full 32 registers of the 32-bit ISA mode, MIPS16e+ contains move instructions to copy values between the eight MIPS16e+ registers and the remaining 24 registers of the full MIPS architecture. Additionally, specific instructions implicitly reference r24 (t8), r28 (gp), r29 (sp), r30 (fp) and r31 (ra). r24 serves as a special condition code register for handling compare results. r28 is the global pointer register. r29 maintains the program stack pointer. r30 is the frame pointer register. r31 is the link register.

■ **Note: Please do not use r1 while programming since r1 is reserved as a register for assembler.**

HI and LO Registers

The HI and LO registers hold the results of integer multiply, divide, multiply-and-add and multiply-and-subtract operations. Integer multiply, multiply-and-add and multiply-and-subtract operations store the doubleword, 64-bit result, in the HI and LO registers. Integer divide operations store the quotient in the LO register and the remainder in the HI register. The MFHI, MFLO, MTHI and MTLO instructions are used to move data between the HI and the LO registers and the general registers.

■ Program Counter (PC)

The least-significant bit of the program counter is the ISA mode bit that determines the ISA mode instructions: 0 means 32-bit ISA and 1 means 16-bit ISA. ISA mode bit is not considered as a part of the address. The address of the on-going instruction is the total value of the entire 32 bit after erasing the least-significant bit.

2.2.2 System Control Coprocessor (CP0) Registers

The system control coprocessor, CP0, is an integral part of the TX19A processor. It has 17 user-accessible registers shown in Figure 2-6.

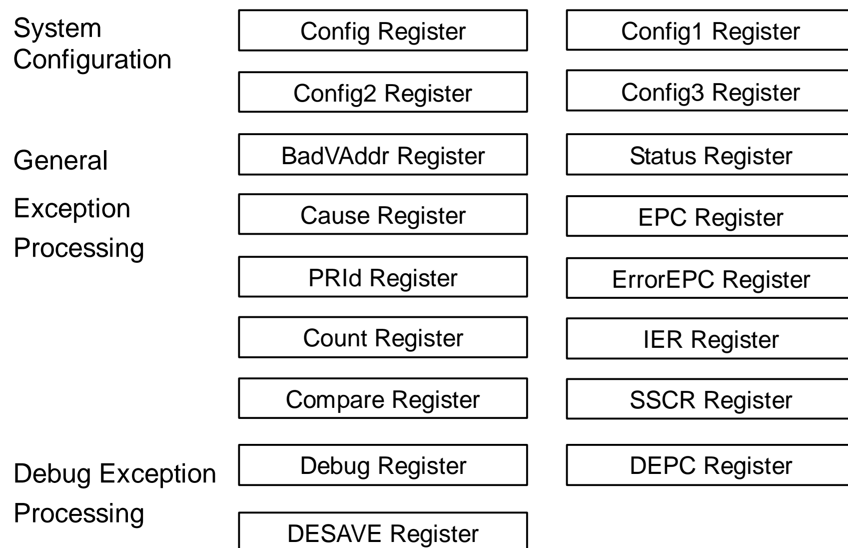


Figure 2-6 System Control Coprocessor (CP0) Registers

The CP0 registers are classified into three groups: system configuration registers, general exception handling registers and debug exception handling registers. When the processor is in Kernel mode, the system control coprocessor instructions can always use the CP0 registers regardless of the setting of the CU0 bit in the Status register. If the processor is in User mode, the CP0 registers are accessible only when the CU0 bit is 1. Operating modes are explained in Section 2.7, *Memory management Summary.*

Table 2-1 System Configuration Register

Register Name	Description
Comfit、Config1、Config2、Config3	System configurations, such as EJTAG and 16-bit ISA mode and cache configurations.

Table 2-2 General Exception Handling Registers

Register Name	Description
BadVAddr	Bad virtual address that caused a virtual-to-physical address translation error. Read-only
Status	Processor status, e.g., operating mode (User/Kernel), interrupt enable and other states.
Cause	Cause of the last exception
EPC	Exception program counter. Upper 31 bits of the address of the exception-causing instruction combined with the ISA mode bit.
ErrorEPC	Similar to the EPC register, except that ErrorEPC is used on Reset and NMI exceptions.
Count	Acts as a timer, incrementing at 1/2 the rate of CPUCLK.
Compare	Maintains a constant value compared against the Count register value.
PRId	Processor revision identifier. Read-only
IER	Manipulates the interrupt enable/disable bit in the Status register.
SSCR	Indicates the previous and current shadow register sets.

Table 2-3 Debug Exception Handling Registers

Register Name	Description
Debug	Cause and current status of a debug exception
DEPC	Debug exception program counter. Upper 31 bits of the address of the instruction that caused a debug exception, combined with the ISA mode bit.
DESAVE	Scratchpad register to save one of the general-purpose registers for context-switching

2.3 32-Bit and 16-Bit ISA Modes

The TX19A has two ISA modes, 16-bit and 32-bit. These operating conditions are respectively called as 16 bit ISA mode and 32 bit ISA mode. It provides an efficient run-time switching between 16-bit and 32-bit ISA modes through an instruction. Programs will consist of procedures in 16-bit mode for density or in 32-bit mode for performance.
The least-significant bit of the program counter (PC) is the ISA mode bit that determines the width of instructions: 0 means 32-bit ISA and 1 means 16-bit ISA. The JALX, JR, JRC or JALRC instructions can be used to switch from 32-bit mode to 16-bit mode or vice versa.

When an exception occurs while the processor is in 16-bit mode, the processor automatically switches to 32-bit mode and saves the return address together with the ISA mode bit to the EPC, ErrorEPC or the DEPC register. The ERET instruction is used to jump back to the return address contained in the EPC or ErrorEPC register. In case of a debug exception, the DERET instruction is used to jump back to the return address contained in the DEPC register.

The instruction set can be divided into the groups shown in Figure 2-7.

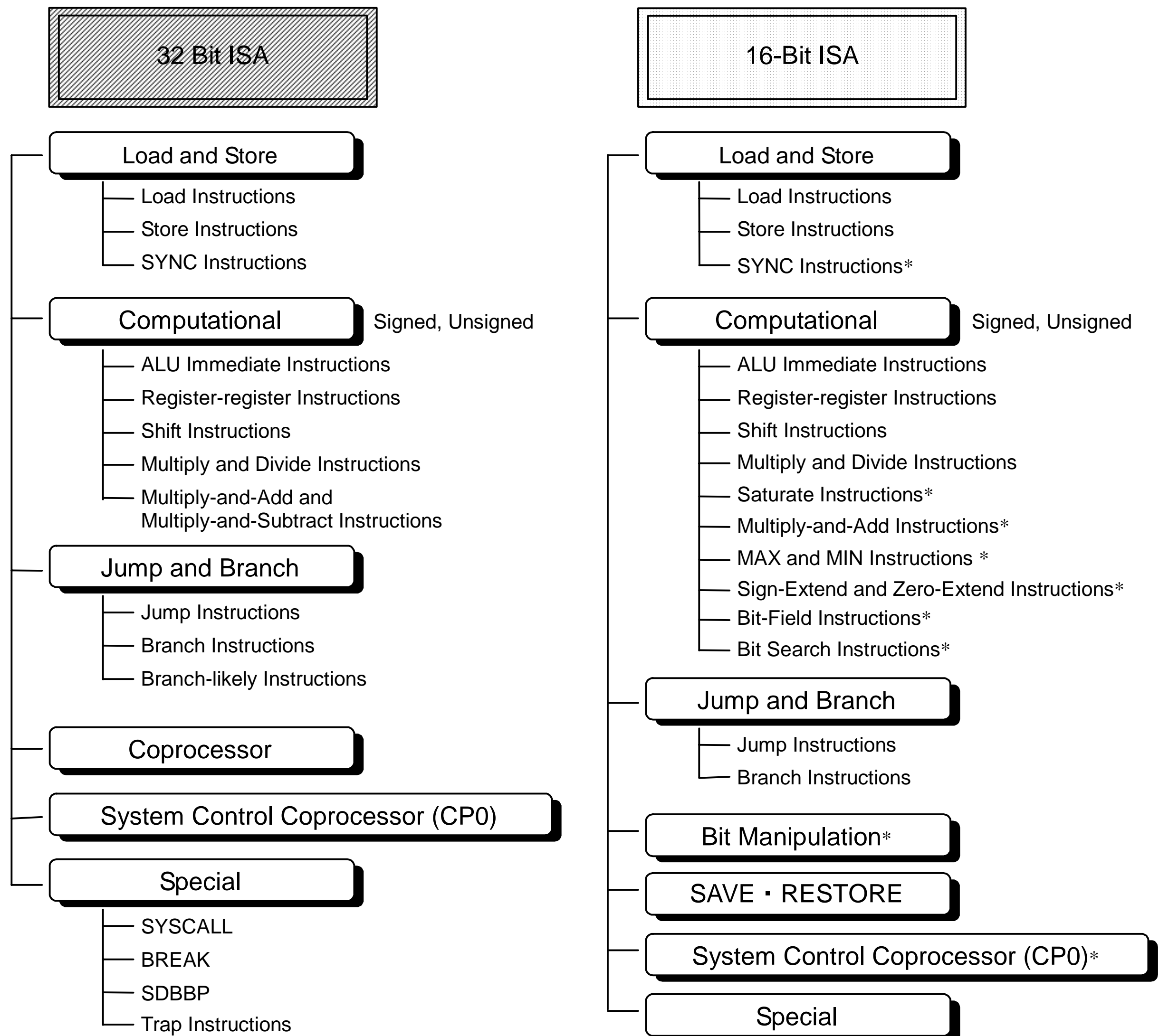


Figure 2-7 32-Bit and 16-Bit Instructions

All the instruction length of 32-bit ISA is set as 32 bit. As a general rule, the instruction length of 16-bit ISA is set as 16 bit; however, it can be changed into 32 bit with a EXTEND instructions. The EXTEND instructions, of which bit size is 16, are consist of 5-bit opcodes and 11-bit immediate. In some cases, the 11-bit immediate field is replaced with an opcode. The EXTEND does not

generate a MIPS machine instruction on its own, but16-bit immediate can be used by concatenating its immediate and an immediate of a subsequent instruction.
The 16-bit ISA instruction with 32-bit instruction length is called EXTENDed instructions. The SYNC, ERET, DERET, WAIT, BS1F, MAX and MIN instructions are EXTENDed instructions and have no 16-bit equivalents.

2.4 Coprocessors

Coprocessors are secondary processors used to speed up operations by handling some of workload of the main CPU.
The TX19A contains a system control coprocessor, CP0, which handles system configuration, exception handling and memory management. The basic capabilities of CP0 are incorporated into the processor core and the extended capabilities into the memory management unit (MMU).
The CU0 bit in the Status register controls the usability of CP0 instructions in User mode. Coprocessor Unusable exception occurs due to CP0 instruction execution during a user-mode program when the CU0 bit is cleared. In Kernel and Debug modes, all CP0 instructions can be executed regardless of the setting of the CU0 bit.
The CU [3:1] bits in the Status register control accesses to the respective coprocessors in User mode or in Kernel mode. Attempted execution of a coprocessor instruction causes a Coprocessor Unusable exception when its CU bit is cleared.
The system control coprocessor (CP0) provides 17 user-visible registers. Chapter 8 gives a complete description of them.

2.5 Pipeline Architecture

The TX19A has a five-stage pipeline. That is, the execution of each instruction consists of five primary stages. Each stage takes approximately one clock cycle; thus the execution of each instruction takes at least five cycles. (The JAL and JALX instructions in the 16-bit ISA mode take longer.) The five-stage pipeline divides the execution of each instruction into five discrete portions and executes up to five instructions simultaneously, as shown in Figure 2-8. The five pipe stages are Fetch (F), Decode (D), Execute (E), Memory Access (M) and Register Write-back (W). The TX19A achieves an instruction execution rate approaching one instruction per clock cycle.

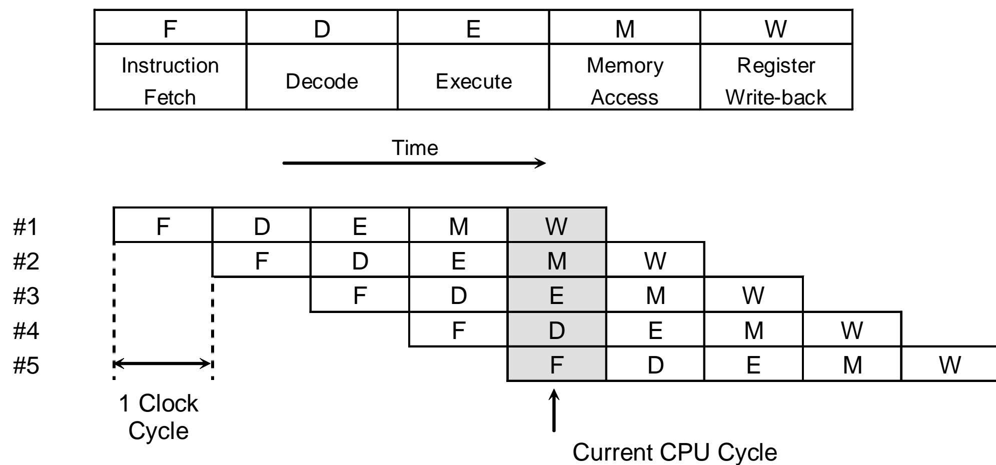


Figure 2-8 TX19A Pipeline

2.6 Write Buffer

A write buffer is a FIFO buffer with 4 entries. As explained in the previous chapter, each pipeline stage takes one clock cycle if the ongoing instruction requires writing areas other than the on-chip memory. Bus cycle for writing to the area other than the on-chip memory not always takes only one clock. The write buffer function can improve performance during the program operation by coordinate the speed differences.

2.6.1 Instructions for Write Buffer

Here are the instructions for the write buffer which generates write bus cycle to memory.

- All the store instructions
  32ISA: SW / SH / SB / SWL / SWR
  16ISA: SW / SH / SB

- A part of bit computational instructions, memory operand addition
  32ISA: none
  16ISA: BCLR / BSET / BINS

- Others
  32ISA: none
  16ISA: ADDMIU / SAVE

Note: Please refer to Appendix A"32-Bit ISA Details" and Appendix B "16-Bit ISA Details" for further details.

2.6.2 Instruction Procedure

At the execution of the instruction to use the write buffer, a bus operation required for executing the instruction is placed in the write buffer. We call it as “entry in the write buffer”. The entry in the write buffer is executed in the order corresponding to instruction execution.

When the write buffer has free space, it enters the bus operation if the instruction to use the write buffer is in the Execute (E) stage. The bus cycle and the entry in the write buffer starts simultaneously if there is no bus cycle executed in operand bus at that time, which means operand bus has free space. When the write buffer has no free space, the instruction stalls in the E stage until it gains appropriate free space.

The earlier the operation is entered in the write buffer, the earlier it is executed when there is a free operand bus. The order will never be changed in the write buffer. The write bus cycle cannot be executed when there is no free operand bus. In case a subsequent instruction such as the LOAD requests the read bus cycle, the instruction stalls in the E stage until all the operations entered in the write buffer are completed.

Figure 2-9 shows the procedure of the write buffer instruction. In this case, the third one is the LOAD instruction. Therefore the read bus cycle caused by the LOAD will not be executed as long as the write cycle during the write buffer operation is completed.

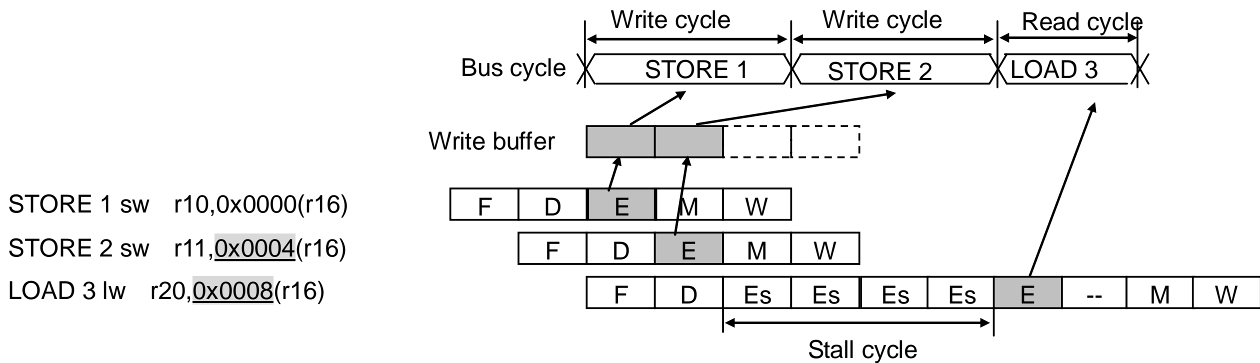


Figure 2-9 The procedure of the write buffer instruction

2.6.3 Bit Computational Instructions/ ADDMIU Instructions

The instructions accompanied by an operand read such as a bit computational or an ADDMIU instructions initiate the operand read bus cycle. The read bus cycle and the write bus cycle are always executed in succession since these cycles must be united as a read modify write operation.

In this case, the write cycle of the bit computation gets priority over the subsequent instructions.

Figure 2-10 shows the procedure of the bit computational instruction.

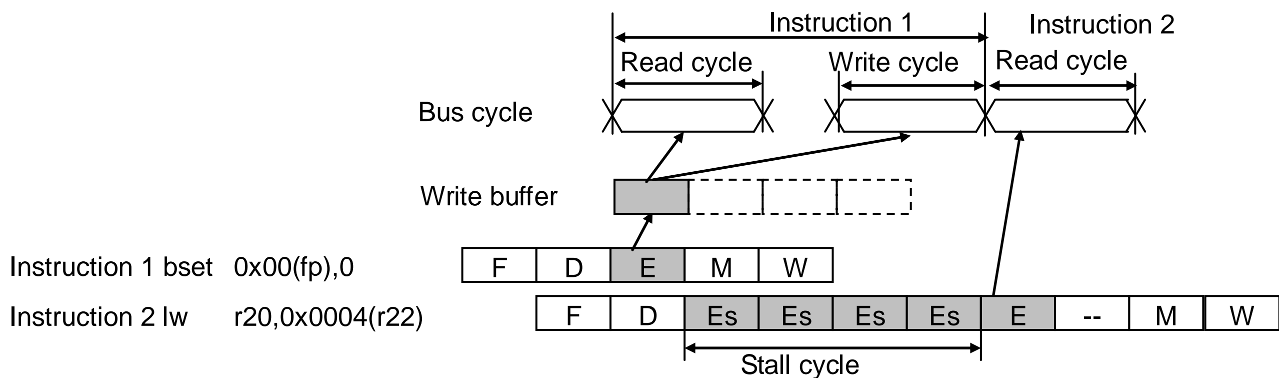


Figure 2-10 The procedure of the bit computational instruction

2.6.4 SAVE Instruction

The SAVE instruction can generate multiple stores. The write buffer starts to enter the save instructions from the earlier store. In the meantime, the SAVE instruction can occupy the execution stage; that is to say no operation caused by other instructions will be entered in the write buffer.

2.6.5 SYNC Instructions

With the SYNC instruction, all the write bus operations entered in the write buffer to maintain the consistency of memory data are executed. The SYNC instruction is effective to synchronize the condition of memory or IO with the instruction operation since this instruction stalls until all the bus cycle caused by the entered operations are completed.

The contents in the write buffer are never automatically flashed when Interrupt/ Exception takes place or bus is opened. Consistency must be maintained by the SYNC instruction depend on the situation.

2.7 Memory Management Summary

The TX19A has two modes of operation, User mode and Kernel mode. The TX19A enters Kernel mode whenever an exception is taken. Since a reset exception occurs when a system is reset, the TX19A wakes up in Kernel mode. The processor switches to User mode when the ERET (Exception Return) or DERET (Debug Exception Return) instruction is executed.

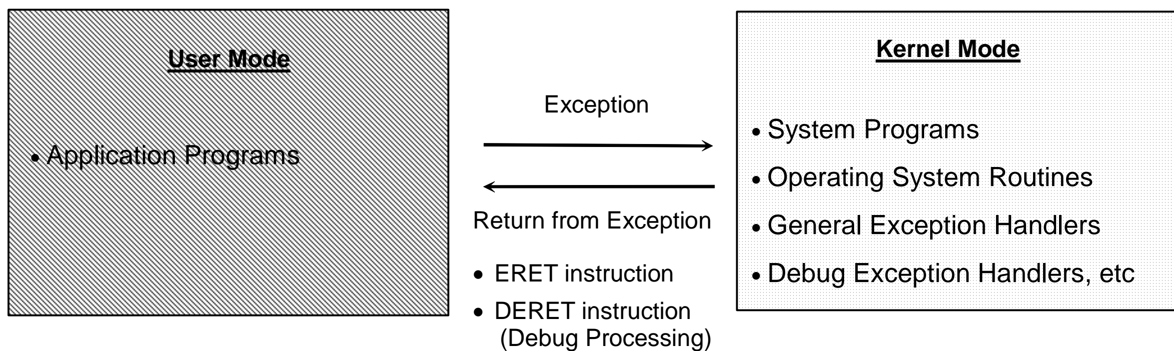


Figure 2-11 Operating Modes

The operating mode determines the addresses, registers and instructions that are available to a program. Kernel mode has higher privileges than User mode. Kernel-mode programs are permitted to use all addresses, registers and instructions, but a User-mode program's use of them are restricted. Operating system routines, general exception handlers and debug exception handlers are executed in Kernel mode. This scheme allows the kernel to protect system resources from uncontrolled access.

Note: TX19A only allows using Kernel mode.

The TX19A does not contain a translation lookaside buffer (TLB). Instead, the memory management unit (MMU) of the TX19A uses the direct segment mapping method. The mapping of virtual addresses to physical addresses is shown in Figure 2-12. The virtual address space is partitioned into four, fixed-size segments. kuseg is designed to be used by User-mode programs while it is accessible in Kernel mode. The other three segments, kseg0, kseg1 and kseg2, are available only to Kernel-mode programs. Chapter 6 describes the memory management features in greater details.

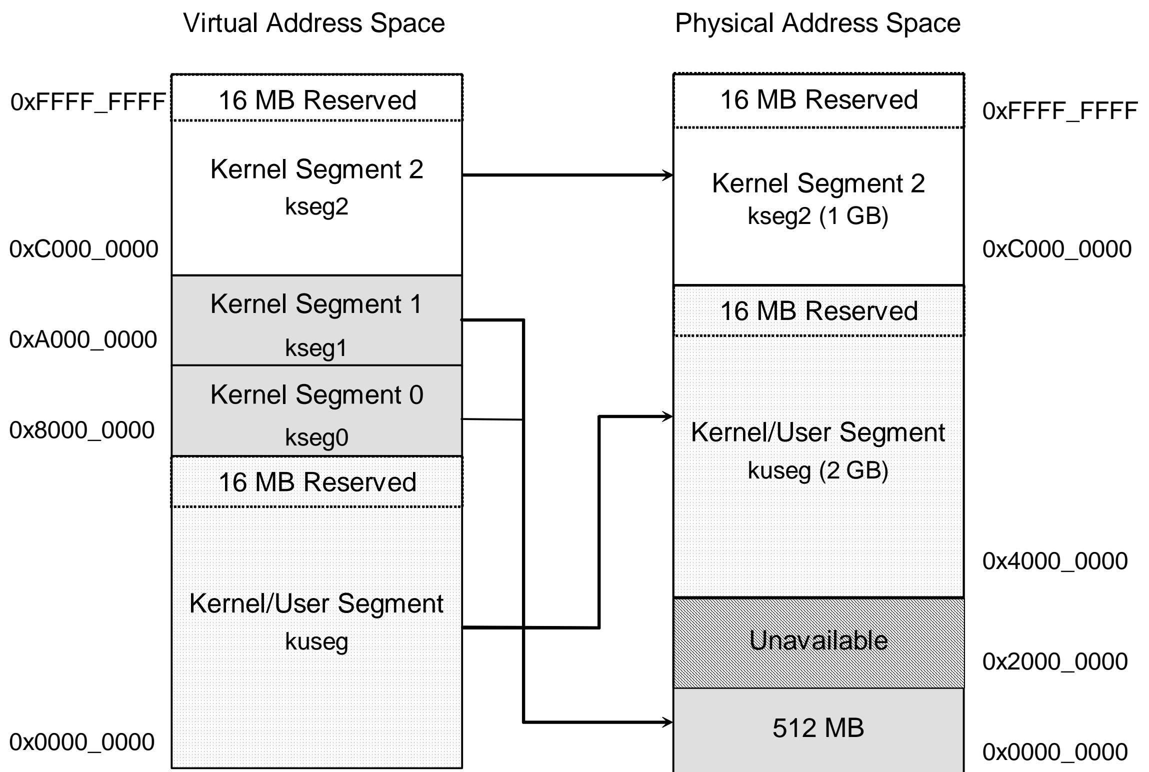


Figure 2-12 Virtual-to-Physical Address Mapping

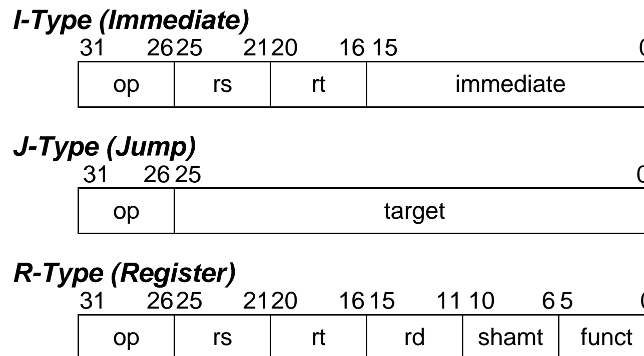
Chapter 3 32-Bit ISA Summary and Programming Tips

This chapter gives an overview of the instructions and addressing modes supported by the TX19A in 32-bit ISA mode. This chapter also presents many programming tips using 32-bit instructions. Instructions are grouped into the following categories:

- Load and store instructions
- Computational instructions
- Jump, branch and branch-likely instructions
- System control coprocessor (CP0) instructions
- Special instructions

3.1 Instruction Formats

All TX19A instructions for the 32-bit ISA mode are 32-bits wide. There are three instruction formats as shown in Figure 3-1. Limiting instruction formats to these three dramatically simplifies instruction decoding. More complex instructions are synthesized by the compiler. All the 32-bit instructions must be aligned on a word boundary.



op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target register specifier or branch condition
immediate	16-bit immediate, or branch or address displacement (offset)
target	26-bit jump target address
rd	5-bit destination register specifier
shamt	5-bit shift amount
funct	6-bit function code

Figure 3-1 Instruction Formats

3.2 Load and Store Instructions

Load and store instructions move data between memory and CPU general registers. Load and store instructions can only load from memory into registers or store registers into memory locations. There is no direct way of doing arithmetic or logical operations between registers and the contents of memory.

3.2.1 Load and Store Address Calculation

In 32-bit ISA mode, all load and store instructions are encoded as I-type instructions. They generate effective addresses using register indirect with offset addressing mode, as shown in Figure 3-2. The 16-bit immediate is sign-extended to 32 bits and added to the contents of a general-purpose register to generate the effective address. For example, in the instruction

```
LW r9,4(r8)
```

4 (binary 0100) is the offset, r8 is a general-purpose register containing the base address, and r9 is the target register.

This addressing mode shown in figure 3-2 can be used to implement immediate addressing using r0 as the base register or register direct addressing using an offset value of zero.

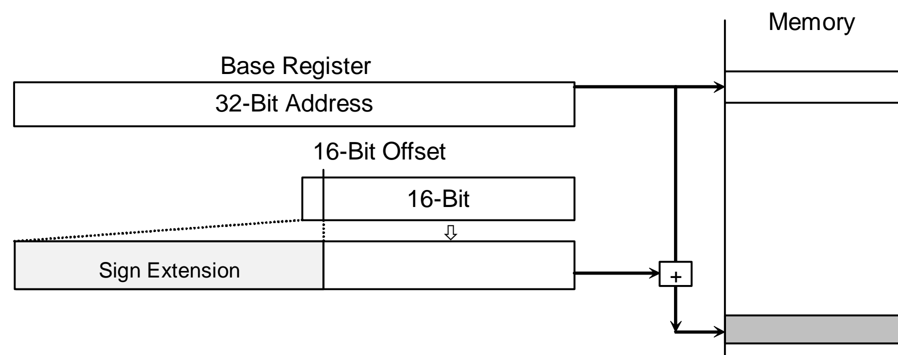


Figure 3-2 Register Indirect with Offset Addressing

3.2.2 Load and Store Instructions for Aligned Accesses

Table 3-1 gives the load and store instructions to perform byte, halfword and word accesses. The LB and LH instructions sign-extend the loaded byte and halfword. The LBU and LHU instructions, which have the “U” (unsigned) suffix, zero-extend the loaded byte and halfword.

Table 3-1 Load and Store Instructions for Aligned Accesses

Data Type	Unsigned Load	Signed Load	Store
Byte	LBU	LB	SB
Halfword	LHU	LH	SH
Word	LW	—	SW

3.2.3 Load and Store Instructions for Misaligned Accesses

An Address Error exception occurs when an instruction to load or store halfword or word that is not aligned on the natural alignment boundary is executed. Table 3-2 gives the instructions to perform loads and stores when the bytes in a word cross the natural boundary between two words. The LWL (Load Word Left) and LWR (Load Word Right) instructions are used in a pair. Likewise, the SWL (Store Word Left) and SWR (Store Word Right) instructions are used in a pair. These instructions provide a more efficient way of dealing with misaligned data than using a sequence of load/store and shift operations. They are useful for reusing old programs written for 8- and 16-bit machines.

Table 3-2 Load and Store Instructions for Misaligned Accesses

	Signed Load	Store
Left (Upper Bytes)	LWL	SWL
Right (Lower Bytes)	LWR	SWR

3.2.4 Memory Synchronization Instruction

The memory synchronization instruction, SYNC, guarantees the sequence of memory references by interlocking the instruction pipeline until loads, stores and instruction fetches performed prior to the present instruction are completed before loads or stores after this instruction are allowed to start.

3.2.5 32-Bit Address Generation

In 32-bit ISA mode, load and store instructions can only take a 16-bit signed immediate as an offset. The most-significant bit is the sign. A total of 15 bits designate the magnitude. This gives a range of -32768 to +32767. If the offset is outside this range, you must put it in a general register prior to the load or store instruction. Three examples are given below.

- Example 1: Base address + 32-bit offset

In the example below, the ADDU (Add Unsigned) instruction is used to add the offset held in register r5 to the base address in register r4. The result is placed back into r4. Then the LW instruction uses r4 as the base register to address a memory location.

```
ADDU    r4,r4,r5
LW      r6,0(r4)
```

- Example 2: Base address + 32-bit offset

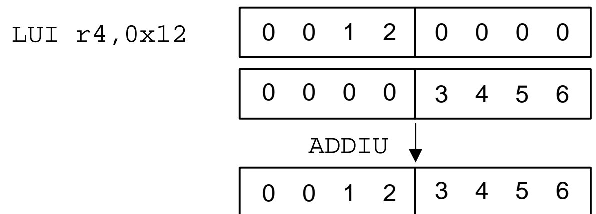
In the example below, the LUI (Load Upper Immediate) instruction loads the 16-bit immediate (in this case, the upper 16 bits of the offset) into the upper 16 bits of register r5. The lower 16 bits of r5 are filled with zeros. Then ADDU (Add Unsigned) instruction is used to add r5 to the base address in r4. This way, the LW instruction can address a desired memory location by only using the lower 16 bits of the offset.

```
LUI     r5,0x12
ADDU    r4,r4,r5
LW      r6,0x3454(r4)
```

- Example 3: Arbitrary 32-bit absolute address

In the example below, the LUI (Load Upper Immediate) instruction loads the 16-bit immediate into the upper 16 bits of register r4. The ADDIU (Add Immediate Unsigned) instruction adds r4 to the lower 16 bits of the offset, 0x3456. The LW instruction can then use r4 to directly address the desired memory location, with an offset of zero.

```
LUI     r4,0x12
ADDIU   r4,r4,0x3456
LW      r6,0(r4)
```



3.3 Computational Instructions

This section describes the computational instructions available in the 32-bit ISA. Section 3.3.1 provides a category of computational instructions. Section 3.3.2 discusses computations that involve the use of 32-bit constants. Section 3.3.3 gives program examples to illustrate how to perform 64-bit addition and subtraction. In Section 3.3.4, we observe how to detect the integer overflow without using exception. In Section 3.3.5, we look at ways to execute a 64-bit x 64-bit multiply operation. Section 3.3.6 describes how to implement rotate operations using available instructions.

3.3.1 Overview of Computational Instructions

Computational instructions in the 32-bit ISA are categorized into five groups shown in Table 3-3. They consist of arithmetic, compare, logical, shift, multiply, divide and multiply-and-add instructions. Computational instructions use I-type format in which one operand is a 16-bit immediate or R-type format which take two or three register operands.

Table 3-3 Computational Instructions

Category	Instructions	Opcode
ALU Immediate	Add	ADDI・ADDIU
	Set On Less Than	SLTI・SLTIU
	Logical AND	ANDI
	Logical OR	ORI
	Logical XOR	XORI
	Load Upper Immediate	LUI
2- and 3-Operand Register-Type	Add	ADD・ADDU
	Subtract	SUB・SUBU
	Set On Less Than	SLT・SLTU
	Logical AND	AND
	Logical OR	OR
	Logical XOR	XOR
	Logical NOR	NOR
	Count	CLO・CLZ
	Conditional Move	MOVN・MOVZ
Shift	Logical Shift	SLL・SLLV・SRL・SRLV
	Arithmetic Shift	SRA・SRAV
Multiply and Divide	Multiply	MULT・MULTU・MUL
	Divide	DIV・DIVU
	Move From/To HI/LO	MFHI・MFLO・MTHI・MTLO
Multiply-and-Add and Multiply-and-Subtract		MADD・MADDU・MSUB・MSUBU

In ALU immediate instructions, the source operands are a general-purpose register and a 16-bit signed immediate. For example, the Add Immediate instruction, "ADDI *rd*, *rs*, *immediate*," adds the contents of the source register (*rs*) and the sign-extended immediate, then places the result into the destination register (*rd*).

Two- and three-operand Register-type instructions manipulate the values held in two general purpose registers and place the result into a general-purpose register.

Shift instructions shift the contents of a general-purpose register right or left by the specified

number of bits. There are two kinds of shift: logical and arithmetic. The Shift Variable instructions (SLLV, SRLV, and SRAV) do not have the shift amount (*shamt*) field; instead they specify a general purpose register containing a desired shift amount.

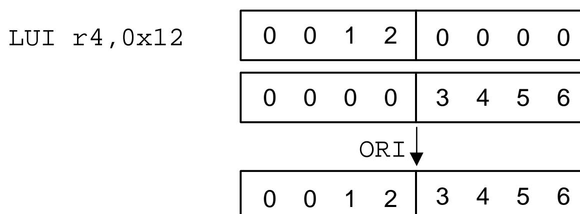
Multiply and divide instructions operate on integer values in two general-purpose registers and place the result into special registers HI and LO. Generally, CPU instructions do not have access to the HI and LO registers. In the MIPS architecture, the MFHI, MFLO, MTHI and MTLO instructions are always required to move data between a general-purpose register and the HI or LO register. However, the TX19A provides an extension to the MIPS architecture to allow the lower 32 bits of the product to be placed into both the LO register and a general-purpose register at a time. Section 3.3.5, *64-Bit x 64-Bit Multiplication*, presents an application example of this extension.

Multiply-and-add and multiply-and-subtract instructions multiply two 32-bit numbers, followed by the addition/subtraction of this product to/from the 64-bit value in the HO/LO registers. The lower 32 bits of the result can be optionally copied into a general-purpose register simultaneously. The MAC unit executes the integer multiply-and-add and multiply-and-subtract operations at an accelerated speed. It is designed to provide a common set of digital signal processing (DSP) operations.

3.3.2 32-Bit Constants

The immediate field in the I-type instructions is only 16-bits long. If the immediate value is greater than 16 bits, you need to use two instructions to create a 32-bit constant and put it in a general register temporarily. In the example below, the LUI (Load Upper Immediate) instruction loads the immediate value into the upper 16 bits of r4 and fills the lower 16 bits with zeros. The ORI (OR Immediate) instruction zero-extends the immediate value, logical-ORs it with the contents of r4 and places the result back into r4.

```
LUI     r4,0x12
ORI     r4,r4,0x3456
```



The following is an example of adding a 32-bit constant to the contents of a general register. The LUI instruction loads the upper 16 bits of r5 with 0x1234 and sets the lower 16 bits to 0x0000. Adding it to 0x5678 with the ADDIU (Add Immediate Unsigned) instruction gives 0x12345678 that is placed back into r5. Finally, the ADDU (Add Unsigned) instruction adds the contents of r4 and r5 together

and puts the result in r6.

```
LUI     r5,0x1234
ADDIU   r5,r5,0x5678
ADDU    r6,r4,r5
```

Note: The ADDI and SLTI instructions sign-extend the immediate value to 32 bits. Although ADDIU and SLTIU stand for Add Immediate *Unsigned* and Set On Less Than Immediate *Unsigned*, they also *sign*-extend the immediate value to 32 bits. The only difference between the ADDI and ADDIU instructions is that ADDIU never causes an overflow exception. Therefore, you can use the ADDIU instruction to add a negative number to the contents of a general register without being worried about a possible overflow. It is useful since there is no Subtract Immediate instruction in the instruction set. The only difference between the SLTI and SLTIU instructions is that SLTI compares two values (*rs* and sign-extended *immediate*) as signed integers while SLTIU compares two values (*rs* and sign-extended *immediate*) as unsigned integers.

3.3.3 64-Bit Addition and Subtraction

In some cases, the numbers being added or subtracted can be more than 32-bits long. Since general purpose registers are only 32-bits wide, it is the job of the programmer (or the compiler) to write the code to break down large numbers into smaller chunks to be processed by the CPU. Figure 3–3 illustrates this. In Figure 3–3, r3 contains the upper 32 bits of a 64-bit constant, and r2 contains the lower 32 bits of that 64-bit constant. Likewise, r5 and r4 together contain a 64-bit constant.

Figure 3-3 64-Bit Addition and Subtraction

Add with Carry

Below is an example of code to add two 64-bit constants together:

```
ADDU r10,r2,r4 # r10 ←r2 + r4
SLTU r11,r10,r2 # r11=1 if r10 (sum) is less than r2
ADD(U) r11,r11,r3 # r11 ←r11 (carry) + r3
ADD(U) r11,r11,r5 # r11 ←r11 + r5
```

The first ADDU instruction adds the lower 32 bits of two constants together and puts the result in r10. The TX19A architecture does not provide a flag bit to indicate whether an arithmetic operation results in a carry-out. Therefore, it is necessary to somehow record an occurrence of a carry-out resulting from an addition. In the case of two positives together, a carry-out occurred if the sum is less than one of the operands added. Then the next SLTU (Set on Less Than Unsigned) instruction sets r11 to 1 if r10 is less than r2. The following two ADD(U) instructions add the carry-out bit (1 or 0) and the upper

32 bits of the two 64-bit constants.

The last two instructions can be either ADD or ADDU. The only difference between these two instructions is that ADDU (Add Unsigned) never causes an integer overflow exception. When you use the ADDU instruction, you need to write the code to explicitly test for an occurrence of the overflow condition. This is discussed in the next section.

■ Subtract with Borrow

In 64-bit subtraction, the code must take care of the borrow of the lower operand. The technique for performing subtract-with-borrow is quite similar to add-with-carry. Below is an example of code to subtract a 64-bit constant from a 64-bit constant.

```
        SLTU r8,r2,r4 # r8=1 if r2 is less than r4
        SUBU r10,r2,r4 # r10 ←r2 - r4
        SUB(U) r11,r3,r5 # r11 ←r3 - r5
        SUB(U) r11,r11,r8 # r11 ←r11 - r8 (borrow)
```

First of all, the SLTU instruction checks if r2 (minuend) is smaller than r4 (subtrahend). If it is, r8 is set to 1. That is, if there is a borrow resulting from the subtraction of the lower 32 bits, its occurrence is recorded in r8. The content of r8 is subtracted in the last SUB(U) instruction.
Again, the only difference between the SUB and SUBU instructions is that SUBU (Subtract Unsigned) never causes an integer overflow exception.

3.3.4 Testing for an Integer Overflow

As explained in the previous section, the signed add and subtract instructions, ADD and SUB, generate an overflow exception if the addition/subtraction resulted in a two's-complement overflow. On the other hand, the unsigned add and subtract instructions, ADDU and SUBU, never cause an overflow exception. If it is necessary to detect signed overflow without using exceptions or to detect overflow for unsigned operations, you need to write a software routine to check for overflow.
It should be observed that, during addition, overflow occurs if the signs of the operands are the same and the sign of the sum is different. Below is an example of code that checks for overflow resulting from signed addition:

```
        ADDU r2,r3,r4 # r2 ←r3 + r4, no exception
        XOR r5,r3,r4 # Compare signs of r3 and r4; if different,
                     # no overflow (r5 < 0)
        BLTZ r5, No_Ov # Branch on less than zero
        XOR r5,r2,r3 # Compare signs of sum and operand; if different,
                     # overflow occurred (r5 < 0)
        BLTZ r5,Ov # Branch on less than zero
No_Ov:
```

During subtraction, overflow occurs if the signs of the operands are not the same and the sign of the remainder is not the same as the sign of the minuend. Below is an example of code that checks for overflow resulting from signed subtraction:

```
        SUBU r2,r3,r4 # r2 ←r3 - r4
        XOR r5,r3,r4 # Compare signs of r3 and r4; if same, no
                     # overflow occurred
        BGEZ r5,No_Ov # Branch on greater than or equal to zero
        XOR r5,r2,r3 # Compare signs of remainder and minuend; if
                     # different, overflow occurred
        BLTZ r5,Ov # Branch on less than zero
No_Ov:
```

3.3.5 64-Bit x 64-Bit Multiplication

In multiplying two integer numbers in the TX19A, they must be in general-purpose registers. In doubleword-by-doubleword multiplication, each 64-bit operand takes two registers since all general purpose registers are only 32-bits wide.
In Figure 3-4, the upper 32 bits of the multiplicand is placed in r3 and the lower 32 bits of it is in r2. Likewise, the multiplier is put in r5 and r4.

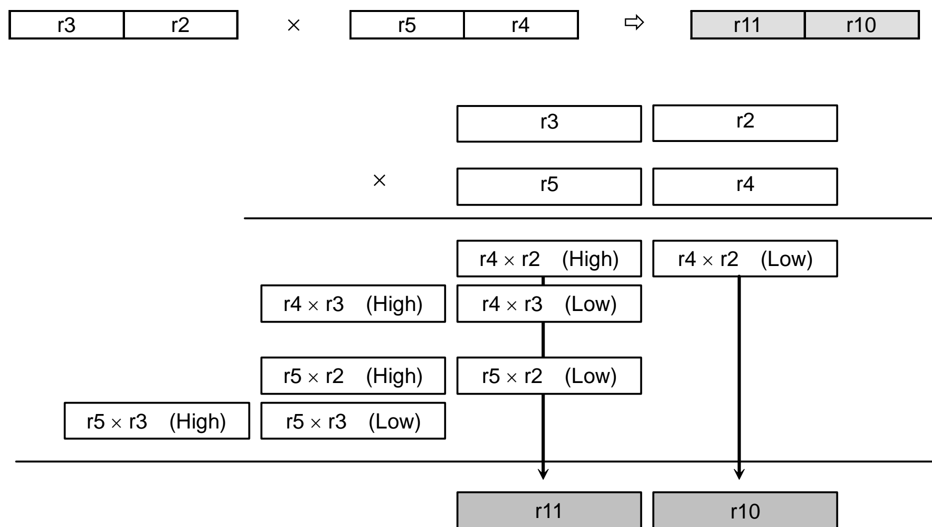


Figure 3-4 64-Bit x 64-Bit Multiplication

The following shows an example of code that performs 64-bit by 64-bit multiplication. Although the product can be a maximum of 128-bits long, the code below only deals with the lower two words of the product for the sake of simplicity.

```
        MULTU r10,r2,r4 # r4 x r2, Copy low word of product to r10
        MFHI r11 # Copy high word of product to r11
        MULTU r9,r3,r4 # r3 x r4, Copy low word of product to r9
        ADDU r11,r11,r9 # r11 ←r11 + r9
```

```
MULTU r9,r2,r5 # r5 x r1, Copy low word of product to r9
ADDU r11,r11,r9 # r11 ←r11 + r9
```

Note that there is a slight difference in the functionality of the MULTU (Multiply Unsigned) instruction between the MIPS and the TX19A architectures. In the MIPS processor, MULTU is a two-operand instruction that specifies two source registers holding the multiplicand and the multiplier. The 64-bit doubleword product is placed into the HI and LO registers. In the TX19A, however, the MULTU can take a third operand. In the TX19A, MULTU can optionally copy the low-order word of the product to a general-purpose register. This eliminates the need to use the MFLO (Move From LO) instruction to move the contents of the LO register to a general register. The MFHI (Move From HI) instruction moves the contents of the HI register, i.e., the high-order word of the product, to a general register.

3.3.6 Rotate Instructions

In the TX19A, there are no rotate instructions at the machine level although it has the shift instructions instead. In shift left, bits that exit the left end (the right end in the case of shift right) are discarded and zeros are supplied to the vacated bits on the right (on the left in the case of shift right). In rotate left, as bits are shifted from right to left (from left to right in the case of rotate right), they exit from the left end, MSB, and enter the right end, LSB, (the left end in the case of rotate right).
In the TX19A, a rotate operation must be implemented using shift and logical-OR instructions. Figure 3-5illustrates how to do this.

```
SLL   r9,r8,6

SRL   r8,r8,(32-6)

OR    r8,r8,r9
```

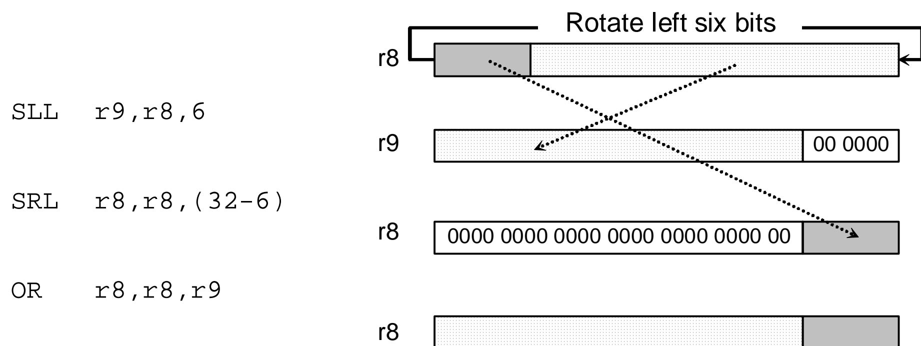


Figure 3-5 Rotate Left by 6 Bits

In Figure 3-5, the SLL (Shift Left Logical) instruction shifts the contents of r8 left by six bits and puts the result in r9. The low-order bits are filled with zeros. Next, the SRL (Shift Right Logical) instruction is used to shift r8 right by 26 (32-6) bits. Finally, the OR instruction logical-ORs the contents of r8 and r9 and puts the result back in r8. The outcome is equivalent to rotating r8 by six bits.

3.4 Jump, Branch and Branch-Likely Instructions

It is often necessary to transfer program control to a different location in the sequence of instructions. There are many instructions to achieve this. The TX19A provides jump, branch and branch-likely instructions. Section 3.4.1 overviews these instructions. Section 3.4.2 describes the addressing modes supported by the jump, branch and branch-likely instructions. Section 3.4.3 explains how to switch from 32-bit ISA mode to 16-bit ISA mode, or vice versa. In Section 3.4.4, the differences between regular branch instructions and branch-likely instructions are explained. Section 3.4.5 provides programming tips for branching on arithmetic comparisons. Section 3.4.6 describes a technique for jumping to 32-bit addresses. Section 3.4.7 describes subroutine calls and returns.

3.4.1 Overview of Jump, Branch and Branch-Likely Instructions

In the TX19A, jump instructions are used to unconditionally transfer program control to the target location whereas branch and branch-likely instructions are what many microprocessors call conditional jumps and are used to transfer control to a new location only when a certain condition is met. Table 3-4 and Table 3-5 show the opcodes of the jump, branch and branch-likely instructions in the 32-bit ISA.

Table 3-4 Jump Instructions (32-Bit ISA)

Opcode	Name	Addressing	Format
J	Jump	Paged absolute	I-type
JAL	Jump And Link	Paged absolute	I-type
JALX	Jump And Link exchange	Paged absolute	I-type
JR	Jump Register	Register indirect	R-type
JALR	Jump And Link Register	Register indirect	R-type

Table 3-5 Branch and Branch-Likely Instructions (32-Bit ISA)

Opcode	Name	Condition	Addressing	Format
B	Unconditional Branch	*always*	PC-relative	I-type
BAL	Branch And Link	*always*	PC-relative	I-type
BEQ(L)	Branch On Equal (Likely)	$rs = rt$	PC-relative	I-type
BNE(L)	Branch On Not Equal (Likely)	$rs \neq rt$	PC-relative	I-type
BGTZ(L)	Branch On Greater Than Zero (Likely)	$rs > 0$	PC-relative	I-type
BGEZ(L)	Branch On Greater Than or Equal To Zero (Likely)	$rs \geq 0$	PC-relative	I-type
BLTZ(L)	Branch On Less Than Zero (Likely)	$rs < 0$	PC-relative	I-type
BLEZ(L)	Branch On Less Than or Equal To Zero (Likely)	$rs \leq 0$	PC-relative	I-type
BLTZAL(L)	Branch On Less Than Zero And Link (Likely)	$rs < 0$	PC-relative	I-type
BGEZAL(L)	Branch On Greater Than or Equal To Zero And Link (Likely)	$rs \geq 0$	PC-relative	I-type

Jump-and-link instructions and branch-and-link instructions save a return address in register r31. They are typically used for subroutine calls.
With the jump and regular branch instructions, the instruction immediately following the jump or branch is always executed while the target instruction is being fetched from memory. This is true to all regular branch instructions regardless of whether the branch is to be taken or not. On the other hand, branch-likely instructions execute the instruction in the delay slot only when the branch is taken; if the branch is not taken, the instruction in the delay slot is nullified. For the jump and branch delay slots, see Chapter 5, *CPU Pipeline*.

3.4.2 Jump and Branch Address Calculation

As shown in Table 3-4 and Table 3-5, jump, branch and branch-likely instructions compute the effective address of the next instruction using the following addressing modes.

- Paged absolute
- Register indirect
- PC-relative with offset

Paged Absolute Addressing

The J, JAL and JALX instructions unconditionally transfer program control to a target address using paged absolute addressing. They generate the next instruction address by shifting the 26-bit immediate operand by two bits and merging the resultant value with the four most-significant bits of the program counter (PC). Figure 3-6 shows how the jump target address is generated by paged absolute addressing. The target address for a jump is computed from the
address of the instruction immediately following the jump instruction, i.e., the address of the jump delay slot. The four most-significant bits of the PC indicate a specific page in a 16-page address space.

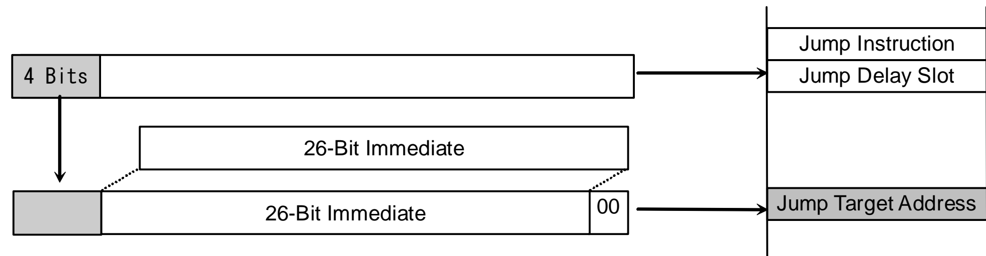


Figure 3-6 Paged Absolute Addressing (32-Bit ISA Mode)

■ Register Indirect Addressing

The JR and JALR instructions unconditionally transfer program control to a target address using a 32-bit absolute address held in a general-purpose register. The effective address is generated by clearing the least-significant bit of the specified target register to zero. Since instructions must be word-aligned, the JR and JALR instructions must specify a target register of which two least-significant bits are zero.

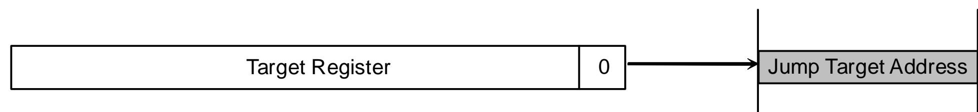


Figure 3-7 Register Indirect Addressing (32-Bit ISA Mode)

■ PC-Relative with Offset Addressing

All the branch and branch-likely instructions transfer program control to a target address using a PC-relative address. They generate the next instruction address by sign-extending and appending b'00 to the 16-bit immediate displacement (offset) operand, and adding the resultant value to the contents of the program counter (PC). Figure 3-8 shows how the branch target address is generated. The target address for a branch is computed from the address of the instruction immediately following the branch instruction, i.e., the address of the branch delay slot.

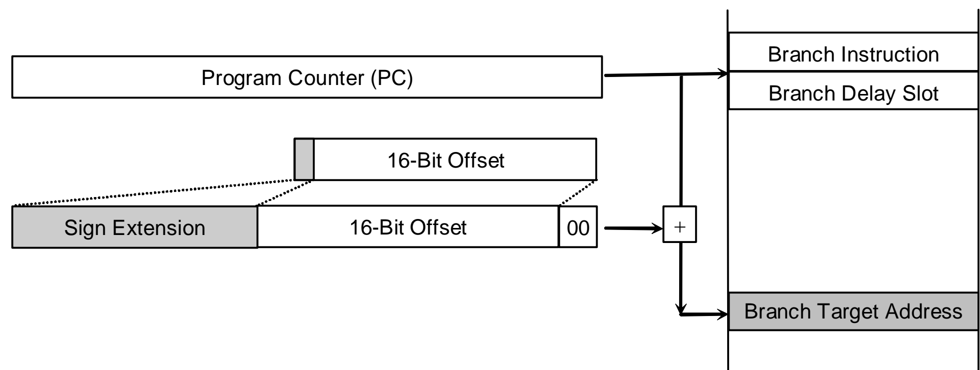


Figure 3-8 PC-Relative with Offset Addressing (32-Bit ISA Mode)

3.4.3 Run-Time Switching of the ISA Modes

The TX19A has two ISA modes, 16-bit ISA and 32-bit ISA. The TX19A provides for efficient runtime switching between 16-bit and 32-bit ISA modes through the JALX, JR and JALR instructions.
The least-significant bit of the program counter (PC) is the ISA mode bit: 0 for the 32-bit ISA and 1 for the 16-bit ISA. The JALX instruction unconditionally toggles the ISA mode bit (the least-significant bit) of the PC to switch to the other ISA. The JR and JALR instructions set the ISA mode bit from the least-significant bit of the register containing the jump address; a jump address is generated by masking off the ISA mode bit to zero.

In 32-bit ISA mode, instructions must be word-aligned. Thus, when switching from 16-bit ISA mode to 32-bit ISA mode, the JR and JALR instructions must specify a target register of which two least-significant bits are zero. If these bits are one-zero (10), an Address Error exception will occur when the jump target instruction is fetched.
In a jump delay slot of the JRLX, JR or JALR instruction, the instruction in the previous ISA mode is executed.

Link instructions save the return address in either register r31 (ra) or another destination register (*rd*) specified. Its least-significant bit keeps the ISA mode in which processing resumes after a subroutine has been executed. Then the same ISA mode as the one prior to the subroutine is set after returning from subroutine.

3.4.4 Branch-Likely Instructions

All the jump and branch instructions occur with a delay of one instruction (two pipeline cycles) before the program flow can change because the processor must calculate the effective destination of the jump or branch and fetch that instruction. This delay is called jump or branch delay. The TX19A architecture gives responsibility of dealing with delay slots to software. The compiler or the assembler makes an attempt to reorder instructions to execute the instruction immediately following the jump or branch while the target instruction is being fetched from memory.

There is no problem in the case of jump instructions since jumps "always" transfer program control to the target instruction; the instruction immediately following the jump can always fill the delay slot. However, with branch instructions, the processor never knows whether the branch will be taken or not; so the instruction in the delay slot must be the one that logically precedes the branch instruction. If the delay slot can not be filled with any useful instruction, a NOP (No Operation) instruction must be inserted to keep the instruction pipeline filled. (NOP is a pseudoinstruction accepted by the assembler; the assembler actually turns it into a shift instruction to r0 register with a shift amount of zero as described in Chapter 1.)
The code in Figure 3-9 implements the task of setting register r2 to 1 or 0, depending on whether the value of r8 is equal to 0 or not. Because the ADDI instruction can not logically precede the BEQ instruction, a NOP instruction is required immediately following BEQ.

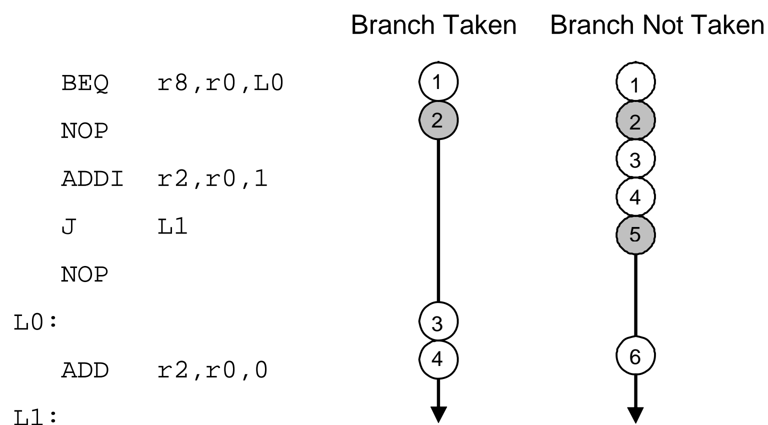


Figure 3-9 Regular Branch Instruction

Contrast this to the code in Figure 3-10 in which the branch-likely version of Branch On Equal (BEQL) is used instead of BEQ. If a branch-likely is taken, the instruction in the delay slot is executed. If a branch-likely is not taken, the instruction in the delay slot is nullified, or killed. This eliminates the need to insert a NOP instruction in the delay slot, and thus helps to reduce code size and speed up branch processing.

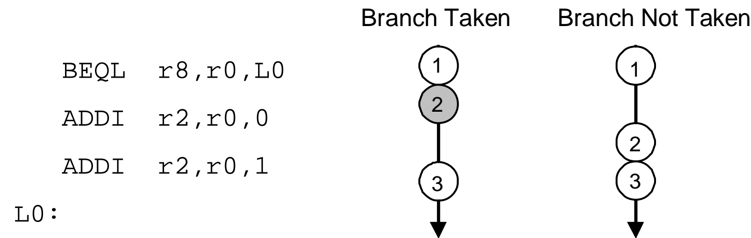


Figure 3-10 Branch-Likely Instruction

3.4.5 Branching on Arithmetic Comparisons

The Branch On Equal (BEQ) and Branch On Not Equal (BNE) instructions, and their branch-likely versions (BEQL/BNEL) are the branch instructions that execute a branch based on the magnitude of two values in registers. For example,

```
BEQ r2,r3,Equal
```

compares the contents of registers r2 and r3 and branches to Equal if they are equal. However, there is no instruction to branch based on whether r2 is greater than r3. To perform such an arithmetic comparison on a pair of registers or between a register and an immediate value, you must use a sequence of two instructions. Three examples are given below: set-on-less-than instructions comparing two registers or a register and an immediate and a comparison between a register and an immediate. (Some assemblers provide macro instructions for branching on arithmetic comparisons. The assembler expands macro instructions into a sequence of machine instructions.)

- Example 1: Branch if r6 ε r7

The following sequence of instructions checks if the contents of r6 is equal to or greater than the contents of r7. If the contents of r6 is less than that of r7, the SLT (Set On Less Than) instruction sets r24 to 1.
Otherwise, r24 is set to 0. The BEQ instruction branches for magnitude relation by detecting r24 value with BEC instruction (Remember r0 is
hardwired to a constant value of zero).

```
SLT    r24,r6,r7
BEQ    r24,r0,Label
```

- Example 2: Branch if r7 ε 0x1234

The following sequence of instructions checks if the contents of r7 is equal to or greater than 0x1234 or not. In this example, the SLTI (Set On Less Than Immediate) instruction is used to compare the contents of a register against an immediate value.

```
SLTI r24,r7,0x1234
BEQ r24,r0,Label
```

- Example 3: Branch if r7 ¯0x1234

The following sequence of instructions checks the equality of the contents of a register and an immediate value. In this example, the ORI (OR Immediate) instruction temporarily loads r10 with 0x1234. Then the BEQ instruction compares the contents of r10 and the contents of r7.

```
ORI r10,r0,0x1234
BEQ r10,r7,Label
```

3.4.6 Jumping to 32-Bit Addresses

As explained in Section 3.4.2, in paged absolute addressing, the J, JAL and JALX instructions can only take a 26-bit immediate. Since it is shifted left by two bits, the address of the target must be within a 256M byte segment. To jump to an arbitrary 32-bit address, load the desired address into a register by using a sequence of the LUI and ORI instructions and then use the JR (Jump Register) instruction. The following code transfers program control to address 0x76543210.

```
LUI r8,0x7654
ORI r8,0x3210
JR r8
```

3.4.7 Subroutine Calls

In the 32-bit ISA, there are Jump-And-Link (JAL, JALX, JALR), Branch-And-Link (BLTZAL, BGEZAL) and Branch-Likely-And-Link (BLTZALL, BGEZALL) instructions. These are typically used as subroutine calls, where the subroutine return address is stored into register r31 (ra). The JALR (Jump-And-Link Register) instruction can use any general-purpose register (*rd*) as the link register.

All the above instructions place the address of the instruction following the delay slot into r31 (ra) or *rd*. Jump-And-Link instructions set the ISA mode in the least-significant bit of r31 or *rd*.

To return from a subroutine, use the JR instruction. The ISA mode bit (i.e., the least-significant bit of the PC) is restored from the least-significant bit of the link register.

When subroutines are nested, the calling subroutine must save the return address in the link register onto the stack before making the call so that it can be overwritten by the callee.

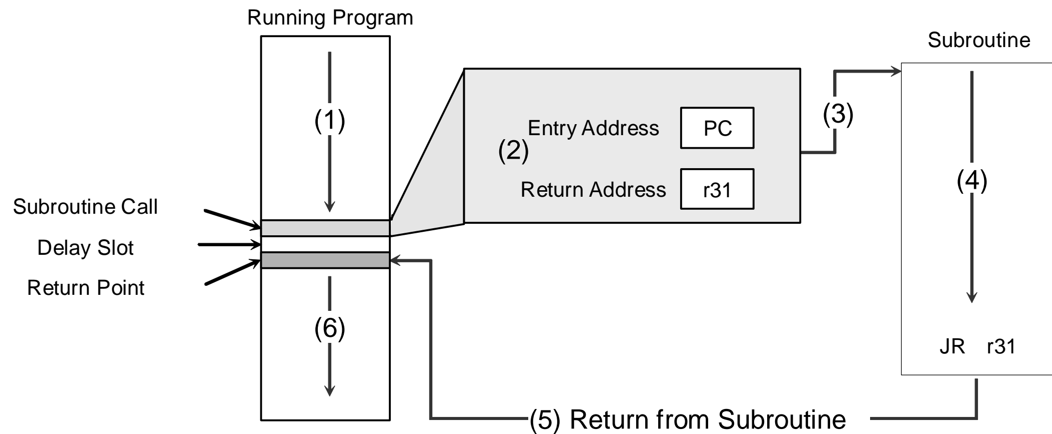


Figure 3-11 Subroutine Calls and Returns

Jump, branch and branch-likely instructions with link except JAL and JALX have a source register (*rs*) field. For example, in the instruction

```
BGEZAL r8,PSUB
```

r8 is the source register; BGEZAL checks if the value in r8 is greater than or equal to zero.

An exception or interrupt could prevent the completion of a legal instruction in the jump or branch delay slot. If that happens, the address of the jump, branch or branch-likely instruction that precedes it is set to the Exception Program Counter (EPC) register. After the exception or interrupt handler routine has been executed, processing restarts with the jump, branch or branch-likely instruction. To permit this, they must be restartable. Therefore, r31 (ra) must not be used as a source register. See Chapter 9 for the exception handling mechanism.

3.5 Coprocessor Instructions

The system control coprocessor (CP0) is implemented as an integral part of the TX19A. No other coprocessor such as CP1 and CP2 can be connected to the TX19A.

Attempts to execute coprocessor instructions (except CP0 instructions) defined in the MIPS32 cause either the Reserved Instruction or Coprocessor Unusable exception. If the corresponding CU bit in the Status register is cleared, a Coprocessor Unusable exception is taken. If the CU bit is set, a Reserved Instruction exception is taken.

The Load Word To Coprocessor (LWC*z*) and Store Word From Coprocessor (SWC*z*) instructions available with the 32 bit ISA are not supported by the TX19A. Attempts to execute these load/store instructions cause a Reserved Instruction exception.

System control coprocessor (CP0) instructions perform operations on the CP0 registers to manipulate the system configuration, memory management and exception handling. Therefore, CP0 is given somewhat protected status. The CU0 bit in the Status register controls the usability of CP0 instructions in User mode. Attempts by a User-mode program to execute a CP0 instruction when the CU0 bit is cleared causes a Coprocessor Unusable exception. In Kernel and Debug modes, all CP0 instructions can be executed, regardless of the setting of the CU0 bit. Table 3-7 shows the CP0 instructions.

Table 3-7 System Control Coprocessor (CP0) Instructions

Name	Opcode
Move To/From CP0	MTC0・MFC0
Exception Return	ERET
Debug Exception Return	DERET
Enter Standby Mode	WAIT

The TX19A performs direct segment mapping of virtual to physical addresses. It does not provide support for a table lookaside buffer (TLB).

3.6 Special Instructions

Special instructions allow software to initiate exceptions, i.e., to test for a particular condition in a running program. All special instructions are R-type. Special instructions include SYSCALL (System Call), BREAK (Breakpoint), SDBBP (Software Debug Breakpoint) and a set of trap instructions. Special instructions transfer program control to an appropriate exception handler. For details on exception processing, see Chapter 9.

Instruction Summary

This section provides an overview of the instructions in the 32-bit ISA.

■ Notational Conventions

In this section, all variable fields in an instruction format are shown in italicized lowercase letters, like *rt*, *rs*, *rd*, *immediate* and *sa* (shift amount). For the sake of clarity, an alias is sometimes used to refer to a field in the formats of specific instructions. For example, *base* and *offset* are used instead of *rs* and *immediate* in the formats of load and store instructions. HI and LO are the special registers that hold the results of integer multiply and divide operations.

■ Extensions

There are several instructions in the TX19A that are not part of the TX19 or TX39 architecture. For a complete list of differences in the instruction set between the TX19A, the TX19 and the TX39, see Appendix D.

Table 3-8 Load and Store Instructions (32-Bit ISA)

Instruction	Format	Operation
Load Byte	LB *rt*, *offset*(*base*)	The effective address is the sum *base* + *offset*. The 16-bit *offset* is sign-extended. The byte in memory addressed by the EA is sign-extended and loaded into *rt*.
Load Byte Unsigned	LBU *rt*, *offset*(*base*)	The effective address is the sum *base* + *offset*. The 16-bit *offset* is sign-extended. The byte in memory addressed by the EA is zero extended and loaded into *rt*.
Load Halfword	LH *rt*, *offset*(*base*)	The effective address is the sum *base* + *offset*. The 16-bit *offset* is sign-extended. The halfword in memory addressed by the EA is sign-extended and loaded into *rt*.
Load Halfword Unsigned	LHU *rt*, *offset*(*base*)	The effective address is the sum *base* + *offset*. The 16-bit *offset* is sign-extended. The halfword in memory addressed by the EA is zero-extended and loaded into *rt*.
Load Word	LW *rt*, *offset*(*base*)	The effective address is the sum *base* + *offset*. The 16-bit *offset* is sign-extended. The word in memory addressed by the EA is loaded into *rt*.
Load Word Left	LWL *rt*, *offset*(*base*)	The effective address is the sum *base* + *offset*. The 16-bit *offset* is sign-extended. The left portion of *rt* is loaded with the appropriate part of the high-order word in memory addressed by the EA.
Load Word Right	LWR *rt*, *offset*(*base*)	The effective address is the sum *base* + *offset*. The 16-bit *offset* is sign-extended. The right portion of *rt* is loaded with the appropriate part of the low-order word in memory addressed by the EA.
Store Byte	SB *rt*, *offset*(*base*)	The effective address is the sum *base* + *offset*. The 16-bit *offset* is sign-extended. The least-significant byte in *rt* is stored in memory addressed by the EA.
Store Halfword	SH *rt*, *offset*(*base*)	The effective address is the sum *base* + *offset*. The 16-bit *offset* is sign-extended. The low-order halfword in *rt* is stored in memory addressed by the EA.
Store Word	SW *rt*, *offset*(*base*)	The effective address is the sum base + offset. The 16-bit offset is sign-extended. *rt* is stored in memory addressed by the EA.
Store Word Left	SWL *rt*, *offset*(*base*)	The effective address is the sum *base* + *offset*. The 16-bit *offset* is sign-extended. The left portion of *rt* is stored into the appropriate part of high-order word of memory addressed by the EA.
Store Word Right	SWR *rt*, *offset*(*base*)	The effective address is the sum *base* + *offset*. The 16-bit *offset* is sign-extended. The right portion of *rt* is stored into the appropriate part of low-order word of memory addressed by the EA.
Sync	SYNC	The instruction pipeline is interlocked until any load or store fetched before the current instruction is completed.

Table 3-9 ALU Immediate Instructions (32-Bit ISA)

Instruction	Format	Operation
Add Immediate	ADDI *rt*, *rs*, *immediate*	The sum *rs* + *immediate* is placed into *rt*. The 16-bit *immediate* is sign-extended. Exceptions on 2's-complement overflow.
Add Immediate Unsigned	ADDIU *rt*, *rs*, *immediate*	The sum *rs* + *immediate* is placed into *rt*. The 16-bit *immediate* is sign-extended. Does not cause exception on 2's-complement overflow.
Set On Less Than Immediate	SLTI *rt*, *rs*, *immediate*	*rt* = 1 if *rs* is less than *immediate*; otherwise *rt* = 0. The 16-bit *immediate* is sign-extended. Two values are compared as signed integers.
Set On Less Than Immediate Unsigned	SLTIU *rt*, *rs*, *immediate*	*rt* = 1 if *rs* is less than *immediate*; otherwise *rt* = 0. The 16-bit *immediate* is sign-extended. Two values are compared as unsigned integers.
AND Immediate	ANDI *rt*, *rs*, *immediate*	The contents of *rs* is ANDed with *immediate* and the result is placed into *rt*. The 16-bit *immediate* is zero-extended.
OR Immediate	ORI *rt*, *rs*, *immediate*	The contents of *rs* is ORed with *immediate* and the result is placed into *rt*. The 16-bit *immediate* is zero-extended.
Exclusive-OR Immediate	XORI *rt*, *rs*, *immediate*	The contents of *rs* is exclusive-ORed with *immediate* and the result is placed into *rt*. The 16-bit *immediate* is zero-extended.
Load Upper Immediate	LUI *rt*, *immediate*	The 16-bit *immediate* is shifted left by 16 bits and concatenated to 16 bits of zeros. The result is placed into *rt*.

Table 3-10 Two- and Three-Operand Register-Type Instructions (32-Bit ISA)

Instruction	Format	Operation
Add	ADD *rd*, *rs*, *rt*	The sum *rs* + *rt* is placed into *rd*. Exceptions on 2's-complement overflow.
Add Unsigned	ADDU *rd*, *rs*, *rt*	The sum *rs* + *rt* is placed into *rd*. Does not cause exception on 2's-complement overflow.
Subtract	SUB *rd*, *rs*, *rt*	The remainder *rs* - *rt* is placed into *rd*. Exceptions on 2's-complement overflow.
Subtract Unsigned	SUBU *rd*, *rs*, *rt*	The remainder *rs* - *rt* is placed into *rd*. Does not cause exception on 2'scomplement overflow.
Set On Less Than	SLT *rd*, *rs*, *rt*	*rd* = 1 if *rs* is less than *rt*; otherwise *rd* = 0. Two values are compared as signed integers.
Set On Less Than Unsigned	SLTU *rd*, *rs*, *rt*	*rd* = 1 if *rs* is less than *rt*; otherwise *rd* = 0. Two values are compared as unsigned integers.
AND	AND *rd*, *rs*, *rt*	The contents of *rs* is ANDed with the contents of *rt* and the result is placed into *rd*.
OR	OR *rd*, *rs*, *rt*	The contents of *rs* is ORed with the contents of *rt* and the result is placed into *rd*.
Exclusive-R	XOR *rd*, *rs*, *rt*	The contents of *rs* is exclusive-ORed with the contents of *rt* and the result is placed into *rd*.
NOR	NOR *rd*, *rs*, *rt*	The contents of *rs* is NORed with the contents of *rt* and the result is placed into *rd*.
* Count Leading Ones in Word	CLO *rd, rs*	*rs* scanned from bit 31 to bit 0. The number of leading ones is counted and the result is placed into *rd*.
* Count Leading Zeros in Word	CLZ *rd, rs*	*rs* scanned from bit 31 to bit 0. The number of leading zeros is counted and the result is placed into *rd*.

* Mover Conditional on Not Zero	MOVN *rd, rs, rt*	If $rt \neq 0$, the contents of *rs* is placed into *rd*.
* Move Conditional on Zero	MOVZ *rd, rs, rt*	If $rt = 0$, the contents of *rs* is placed into *rd*.

* Enhancements from the TX19 to the TX19A

Table 3-11 Shift Instructions (32-Bit ISA)

Instruction	Format	Operation
Shift Left Logical	SLL *rd*, *rt*, *sa*	The contents of *rt* is shifted left by *sa* bits. Zeros are supplied to the vacated positions on the right. The result is placed into *rd*.
Shift Left Logical Variable	SLLV *rd*, *rt*, *rs*	The contents of *rt* is shifted left by the number of bits specified by the five least-significant bits of *rs*. Zeros are supplied to the vacated positions on the right. The result is placed into *rd*.
Shift Right Logical	SRL *rd*, *rt*, *sa*	The contents of *rt* is shifted right by *sa* bits. Zeros are supplied to the vacated positions on the left. The result is placed into *rd*.
Shift Right Logical Variable	SRLV *rd*, *rt*, *rs*	The contents of *rt* is shifted right by the number of bits specified by the five least-significant bits of *rs*. Zeros are supplied to the vacated positions on the left. The result is placed into *rd*.
Shift Right Arithmetic	SRA *rd*, *rt*, *sa*	The contents of *rt* is shifted right by *sa* bits. The sign bit is copied to the vacated positions on the left. The result is placed into *rd*.
Shift Right Arithmetic Variable	SRAV *rd*, *rt*, *rs*	The contents of *rt* is shifted right by the number of bits specified by the five least-significant bits of *rs*. The sign bit is copied to the vacated positions on the left. The result is placed into *rd*.

Table 3-12 Multiply and Divide Instructions (32-Bit ISA)

Instruction	Format	Operation
* Multiply	MUL *rd*, *rs*, *rt*	The multiplicand is the signed value of *rs*. The multiplier is the signed value of *rt*. The low-order 32 bits of the product is placed into *rd*. The values of registers HI and LO become undefined.
Multiply	MULT (*rd*,) *rs*, *rt*	The multiplicand is the signed value of *rs*. The multiplier is the signed value of *rt*. The 64-bit product *rs* * *rt* is placed into registers HI and LO. The low-order 32 bits of the product can be optionally copied into *rd*.
Multiply Unsigned	MULTU (*rd*,) *rs*, *rt*	The multiplicand is the unsigned value of *rs*. The multiplier is the unsigned value of *rt*. The 64-bit product *rs* * *rt* is placed into registers HI and LO. The low-order 32 bits of the product can be optionally copied into *rd*.
Divide	DIV *rs*, *rt*	The dividend is the signed value of *rs*. The divisor is the signed value of *rt*. The quotient is placed into register LO and the remainder is placed into register HI.
Divide Unsigned	DIVU *rs*, *rt*	The dividend is the unsigned value of *rs*. The divisor is the unsigned value of *rt*. The quotient is placed into register LO and the remainder is placed into register HI.
Move From HI	MFHI *rd*	The contents of register HI is copied to *rd*.
Move From LO	MFLO *rd*	The contents of register LO is copied to *rd*.
Move To HI	MTHI *rs*	The contents of *rs* is copied to register HI.
Move To LO	MTLO *rs*	The contents of *rs* is copied to register LO.

* Enhancement from the TX19 to the TX19A

Table 3-13 Multiply-and-Add Instructions (32-Bit ISA)

Instruction	Format	Operation
Multiply and Add	MADD (*rd*,) *rs*, *rt*	The multiplicand is the signed value of *rs*. The multiplier is the signed value of *rt*. The 64-bit product *rs* * *rt* is added to the contents of registers HI and LO and the result is placed back into HI and LO. The low-order 32 bits of the result can be optionally copied to *rd*.
Multiply and Add Unsigned	MADDU (*rd*,) *rs*, *rt*	The multiplicand is the unsigned value of *rs*. The multiplier is the unsigned value of *rt*. The 64-bit product *rs* * *rt* is added to the contents of registers HI and LO and the result is placed back into HI and LO. The low-order 32 bits of the result can be optionally copied to *rd*.
* Multiply and Subtract	MSUB (*rd*,) *rs*, *rt*	The multiplicand is the signed value of *rs*. The multiplier is the signed value of *rt*. The 64-bit product *rs* * *rt* is subtracted from the contents of registers HI and LO and the result is placed back into HI and LO. The low-order 32 bits of the result can be optionally copied to *rd*.
* Multiply and Subtract Unsigned	MSUBU (*rd*,) *rs*, *rt*	The multiplicand is the unsigned value of *rs*. The multiplier is the unsigned value of *rt*. The 64-bit product *rs* * *rt* is subtracted from the contents of registers HI and LO and the result is placed back into HI and LO. The low-order 32 bits of the result can be optionally copied to *rd*.

* Enhancements from the TX19 to the TX19A

Table 3-14 Jump Instructions (32-Bit ISA)

Instruction	Format	Operation
Jump	J *target*	The program jumps to the address computed using paged absolute addressing, i.e., by shifting the 26-bit *target* left by two bits and combining it with the four most-significant bits of PC + 4.
Jump And Link	JAL *target*	The program jumps to the address computed using paged absolute addressing, i.e., by shifting the 26-bit *target* left by two bits and combining it with the four most-significant bits of PC + 4. The address of the instruction following the delay slot is saved in r31.
Jump And Link exchange	JALX *target*	The program jumps to the address using paged absolute addressing, i.e., by shifting the 26-bit *target* left by two bits and combining it with the four most-significant bits of PC + 4. The address of the instruction following the delay slot is saved in r31. The ISA mode bit in the PC toggles.
Jump Register	JR *rs*	The program jumps to the address specified by *rs*, with the least-significant bit cleared. The least-significant bit of *rs* is interpreted as the ISA mode specifier.
Jump And Link Register	JALR (*rd*,) *rs*	The program jumps to the address specified by *rs*, with the least-significant bit cleared. The least-significant bit of *rs* is interpreted as the ISA mode specifier. The address of the instruction following the delay slot is saved in *rd*. If *rd* is omitted, the default is r31.

Table 3-15 Branch and Branch-Likely Instructions (32-Bit ISA)

Instruction	Format	Operation
Branch On Equal (Likely)	BEQ(L) *rs*, *rt*, *offset*	If *rs* = *rt,* the program branches to the target address specified as a 16-bit *offset* relative to PC + 4 (i.e., the address of the branch delay slot).
Branch On Not Equal (Likely)	BNE(L) *rs*, *rt*, *offset*	If *rs* ≠ *rt,* the program branches to the target address specified as a 16-bit *offset* relative to PC + 4 (i.e., the address of the branch delay slot).
Branch On Greater Than Zero (Likely)	BGTZ(L) *rs*, *offset*	If *rs* > 0, the program branches to the target address specified as a 16-bit *offset* relative to PC + 4 (i.e., the address of the branch delay slot).
Branch On Greater Than or Equal to Zero (Likely)	BGEZ(L) *rs*, *offset*	If *rs* ≥ 0, the program branches to the target address specified as a 16-bit *offset* relative to PC + 4 (i.e., the address of the branch delay slot).
Branch On Less Than Zero (Likely)	BLTZ(L) *rs*, *offset*	If *rs* < 0, the program branches to the target address specified as a 16-bit *offset* relative to PC + 4 (i.e., the address of the branch delay slot).
Branch On Less Than or Equal to Zero (Likely)	BLEZ(L) *rs*, *offset*	If *rs* ≤ 0, the program branches to the target address specified as a 16-bit *offset* relative to PC + 4 (i.e., the address of the branch delay slot).
Branch On Less Than Zero And Link (Likely)	BLTZAL(L) *rs*, *offset*	If *rs* < 0, the program branches to the target address specified as a 16-bit *offset* relative to PC + 4 (i.e., the address of the branch delay slot). The address of the instruction following the delay slot is saved in r31.
Branch On Greater Than or Equal to Zero And Link (Likely)	BGEZAL(L) *rs*, *offset*	If *rs* ≥ 0, the program branches to the target address specified as a 16-bit *offset* relative to PC + 4 (i.e., the address of the branch delay slot). The address of the instruction following the delay slot is saved in r31.
* Unconditional Branch	B *offset*	The program unconditionally branches to the target address specified as a 16-bit *offset* relative to PC + 4 (i.e., the address of the branch delay slot).
* Branch And Link	BAL *offset*	The program unconditionally branches to the target address specified as a 16-bit *offset* relative to PC + 4 (i.e., the address of the branch delay slot). The address of the instruction following the delay slot is saved in r31.

* Enhancements from the TX19 to the TX19A

† The "L" suffix in the opcodes indicates a branch-likely instruction.

Table 3-16 System Control Coprocessor (CP0) Instructions (32-Bit ISA)

Instruction	Format	Operation
Move To CP0	MTC0 *rt*, *rd*	The contents of general register *rt* is copied into CP0 register *rd.*
Move From CP0	MFC0 *rt*, *rd*	The contents of CP0 register *rt* is copied into general register *rd.*
* Exception Return	ERET	If the ERL bit in the Status register is 1, the processor returns from an exception and then program execution continues at the address held in the Error EPC register. If the ERL bit is 0, the processor returns from an exception and then program execution continues at the address held in the EPC register.
Debug Exception Return	DERET	Program control is transferred back to a User program from a debug exception handler. The return address in the DEPC register is restored into the PC.
* Enter Standby Mode	WAIT	The processor enters either HALT or DOZE mode, depending on the setting of the PR bit in the Status register.

* Enhancements from the TX19 to the TX19A

Table 3-17 Special Instructions (32-Bit ISA)

Instruction	Format	Operation
System Call	SYSCALL *code*	A system call exception occurs, immediately and unconditionally transferring control to the exception handler.
Breakpoint	BREAK *code*	A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.
Software Debug Breakpoint Exception	SDBBP *code*	A debug breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.
Trap If Equal	TEQ *rs*, *rt*	If *rs* = *rt*, a Trap exception occurs.
* Trap If Equal Immediate	TEQI *rs*, *immediate*	If *rs* = *immediate*, a Trap exception occurs. The 16-bit *immediate* is sign-extended. Two values are compared as signed integers.
* Trap If Greater Than or Equal	TGE *rs, rt*	If *rs* ≥ *rt*, a Trap exception occurs. Two values are compared as signed integers.
* Trap If Greater Than or Equal Immediate	TGEI *rs, immediate*	If *rs* ≥ *immediate*, a Trap exception occurs. The 16-bit *immediate* is sign-extended. Two values are compared as signed integers.
* Trap If Greater Than or Equal Immediate Unsigned	TGEIU *rs, immediate*	If *rs* ε *immediate*, a Trap exception occurs. The 16-bit *immediate* is sign-extended. Two values are compared as unsigned integers.
* Trap If Greater Than or Equal Unsigned	TGEU *rs, rt*	If *rs* ≥ *rt*, a Trap exception occurs. Two values are compared as unsigned integers.
* Trap If Less Than	TLT *rs, rt*	If *rs* < *rt*, a Trap exception occurs. Two values are compared as signed integers.
* Trap If Less Than Immediate	TLTI *rs, immediate*	If *rs* < *immediate*, a Trap exception occurs. The 16-bit *immediate* is sign-extended. Two values are compared as signed integers.
* Trap If Less Than Immediate Unsigned	TLTIU *rs, immediate*	If *rs* < *immediate*, a Trap exception occurs. The 16-bit *immediate* is sign-extended. Two values are compared as unsigned integers.
* Trap If Less Than Unsigned	TLTU *rs, rt*	If *rs* < *rt*, a Trap exception occurs. Two values are compared as unsigned integers.
* Trap If Not Equal	TNE *rs, rt*	If *rs* ≠ *rt*, a Trap exception occurs.
* Trap If Not Equal Immediate	TNEI *rs, immediate*	If *rs* ≠ *immediate*, a Trap exception occurs. The 16-bit *immediate* is sign-extended. Two values are compared as signed integers.

* Enhancements from the TX19 to the TX19A

Chapter 4 16-Bit ISA Summary and Programming Tips

This chapter gives an overview of the instructions and addressing modes supported by the TX19A in 16-bit ISA mode. This chapter also presents many programming tips using 16-bit ISA instructions. Instructions are grouped into the following categories. Branch-likely instructions are not supported by the 16-bit ISA.

- Load and store instructions
- Computational instructions
- Jump and branch instructions
- Bit manipulation instructions
- SAVE and RESTORE instructions
- System control coprocessor (CP0) instructions
- Special instructions

Doubleword instructions available in the MIPS16 ASE are not implemented in the TX19A.

To the 16-bit ISA, only eight of the 32 general-purpose registers are normally visible, r2 to r7, r16 and r17. Since the processor includes the full 32 registers of the 32-bit ISA mode, the 16-bit ISA includes MOVE instructions to copy values between the eight 16-bit-ISA registers and the remaining 24 registers of the full 32-bit architecture. Additionally, specific instructions implicitly reference r24 (t8), r28 (gp), r29 (sp), r30 (fp) and r31 (ra). r24 serves as a special condition code register for handling compare results. r28 is the global pointer register. r29 maintains the program stack pointer. r30 is the frame pointer register. r31 is the link register. Multiply and divide instructions use the special registers HI and LO.

4.1 Instruction Formats

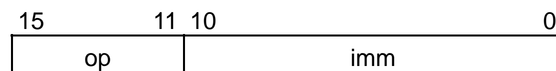
There are 21 instruction formats shown in Figure 4-1 for the 16-bit instructions. There are 20 instruction formats shown in Figure 4-2 for the 32-bit instructions.

To fit within the 16-bit limit, immediate fields in the 16-bit instructions are only 3 to 11 bits. Thus, the 16-bit ISA provides a way to extend its shorter immediates into the full width of immediates in the 32-bit ISA mode. The EXTEND instruction in the 16-bit ISA is not really an instruction and does not generate a machine instruction on its own. It provides a prefix to be prepended to any 16-bit instruction with an address or immediate field. Therefore, EXTENDing typical 16-bit instructions to 32 bits gives several more instruction formats shown in Figure 4-2. For example, the EXTENDed version of the I-type format is called EXT-I.
Additionally, the 16-bit ISA has several 32-bit instructions prepended with the EXTEND code. In such instructions, the 11-bit immediate field in the EXTEND code is replaced with an opcode.

op	5-bit operation code
rx	3-bit source/destination register specifier
ry	3-bit source/destination register specifier
immediate, imm or ximm3	3-, 4-, 5-, 8- or 11-bit immediate, or branch or address displacement (offset)
rz	3-bit source/destination register specifier
F	1-, 2-, 3- or 5-bit function code
r32	32-bit ISA general-purpose register specifier
ra	r31 register
s0	r16 register
s1	r17 register
pos3	Bit number of a specific bit of a memory byte
cpr32	Coprocessor register
hase	fp, sp, gp or r0 register
xsregs	Registers saved or restored
aregs	Registers saved or restored
framesize	Size of frame required

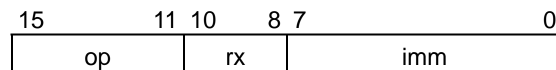
<< **16-Bit Instructions** >>

I Type



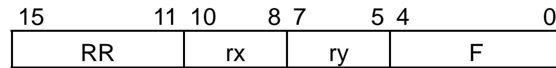
op: B

RI Type

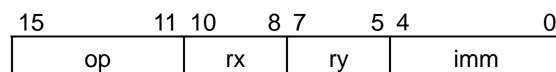


op: ADDIU8・ADDIUPC・ADDIUSP・BEQZ・BNEZ・CMPI・LI・LWPC・LWSP・SLTI・SLTIU SWSP

RR Type

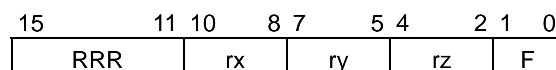


RRI Type

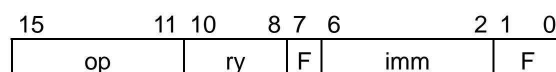


op: LB・LBU・LH・LHU・LW・SB・SH・SW

RRR Type 1

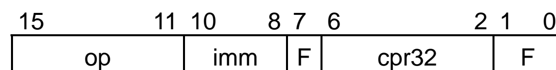


RRR Type 2



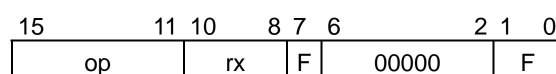
op: SLL・SRL・SRA

RRR Type 3



op: AC0IU

RRR Type 4



op: MTHI・MTLO

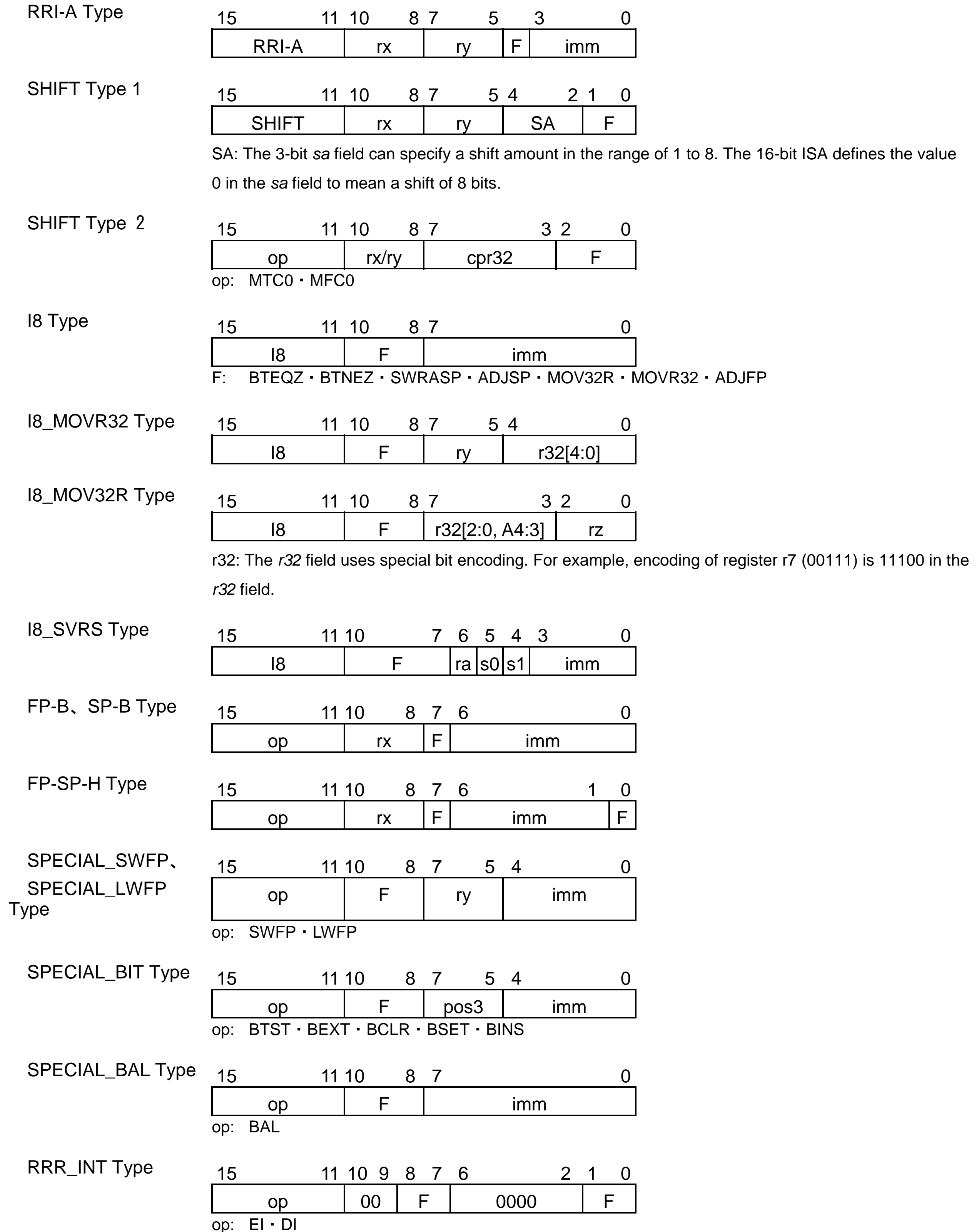


Figure 4-1 16-Bit Instruction Formats

<< **32-Bit Instructions** >>

JAL・JALX Type

31 27	26	25 21	20 16	15 0
JAL	X	TAR[20:16]	TAR[25:21]	TAR[15:0]

X=0: JAL instruction, X=1: JALX instruction

EXT-I Type

31 27	26 21	20 16	15 11	10	9	8	7	6	5	4 0
EXTEND	imm[10:5]	imm[15:11]	op	0	0	0	0	0	0	imm[4:0]

EXT-RI Type

31 27	26 21	20 16	15 11	10 8	7	6	5	4 0
EXTEND	imm[10:5]	imm[15:11]	op	rx	0	0	0	imm[4:0]

EXT-RRI Type

31 27	26 21	20 16	15 11	10 8	7 5	4 0
EXTEND	imm[10:5]	imm[15:11]	op	rx	ry	imm[4:0]

EXT-RRI-A Type

31 27	26 20	19 16	15 11	10 8	7 5	4	3 0
EXTEND	imm[10:4]	imm[14:11]	RRI-A	rx	ry	F	imm[3:0]

EXT-SHIFT Type

31 27	26 22	21	20	19	18	17	16	15 11	10 8	7 5	4	3	2	1 0
EXTEND	SA[4:0]	0	0	0	0	0	0	SHIFT	rx	ry	0	0	0	F

EXT-I8 Type

31 27	26 21	20 16	15 11	10 8	7	6	5	4 0
EXTEND	imm[10:5]	imm[15:11]	I8	F	0	0	0	imm[4:0]

EXT-FP-B、EXT-SP-B Type

31 27	26 21	20 16	15 11	10 8	7	6 5	4 0
EXTEND	imm[10:5]	imm[15:11]	op	rx	F	00	imm[4:0]

EXT-FP-SP-H Type

31 27	26 21	20 16	15 11	10 8	7	6 5	4 1	0
EXTEND	imm[10:5]	imm[15:11]	op	rx	F	00	imm[4:1]	F

EXT-SPECIAL-SWFP, EXT-SPECIAL-LWFP Type

31 27	26 21	20 16	15 11	10 8	7 5	4 0
EXTEND	imm[10:5]	imm[15:11]	op	F	ry	imm[4:0]

EXT-SPECIAL-BIT Type

31 27	26 21	20 19	18 16	15 11	10 8	7 5	4 0
EXTEND	imm[10:5]	base	imm[13:11]	op	F	pos3	imm[4:0]

EXT-SPECIAL- BAL Type

31 27	26 21	20 16	15 11	10 8	7 5	4 0
EXTEND	imm[10:5]	imm[15:11]	op	F	000	imm[4:0]

EXT-ADDIU8 Type

31 27	26 21	20 16	15 11	10 8	7 5	4 0
EXTEND	imm[10:5]	imm[15:11]	op	ry	F	imm[4:0]

op: ANDI・ORI・XORI・LUI

EXT-ADDMIU Type

31 27	26 21	20 19	18 16	15 11	10 8	7 5	4 0
EXTEND	imm[10:5]	base	imm[13:11]	op	ximm3	F	imm[4:0]

EXT-I8-SVRS Type

31 27	26 24	23 20	19 16	15 11	10 8	7	6	5	4	0
EXTEND	xsregs	framesize	aregs	I8	SVRS	F	ra	s0	s1	framesize

EXT-RR Type

31 27	26	25	24 16	15 11	10 5	4 0
EXTEND	0	1	000000000	11101	000000	op2

op2: ERET・DERET・WAIT

EXT-RR-SYSCALL Type

31 27	26 22	21 16	15 11	10 5	4 0
EXTEND	imm[10:6]	imm[16:11]	11101	000000	01100

EXT-RR-BSIF Type

31 27	26	25 16	15 11	10 8	7 5	4 0
EXTEND	1	0000000000	op	ry	rx	00111

EXT-RR-BFINS Type

31 27	26	25 21	20 16	15 11	10 8	7 5	4 0
EXTEND	0	bit2	bit1	op	ry	rx	00111

EXT-RR-MAX/MIN Type

31 27	26	25 24	23 19	18 16	15 11	10 8	7 5	4 0
EXTEND	M	00	00000	ry	op	rz	rx	00101

M=0: MAX Instruction, M=1: MIN Instruction

Figure 4-2 32-Bit Instruction Formats

4.2 Load and Store Instructions

In the 16-bit ISA, there are no load/store instructions for misaligned data. In the 16-bit ISA, the biggest saving in the instruction length comes from restrictions on the size of immediate values expressible. All 16-bit load and store instructions are restricted to 5 to 8 bits of unsigned values. To overcome this restriction, the 16-bit ISA contains a mechanism to EXTEND an address or offset field to 16 bits. For details on the EXTEND instruction, see Section 4.5, *Special Instructions*. To further address the supply of constants, the 16-bit ISA has new addressing modes.

Section 4.2.1 describes the addressing modes supported by the 16-bit load and store instructions. Section 4.2.2 gives an overview of the load and store instructions. Section 4.2.3 explains how to get 32-bit addresses using an addressing mode newly added to TX19A. Section 4.2.4 describes the SYNC instruction.

4.2.1 Load and Store Address Calculation

In the 16-bit ISA, there are four addressing modes supported by load and store instructions:

- Register indirect with offset
- SP-relative with offset
- FP-relative with offset
- PC-relative with offset

Register Indirect with Offset Addressing

In 16-bit ISA mode, most load and store instructions use register indirect with offset addressing. Instructions using this addressing mode is the RRI (register-register-immediate) type and include a base register and an unsigned 5-bit offset field. These instructions generate the target address by zero-extending the 5-bit offset and adding it to the contents of the base register. The base register can be any of the general-purpose registers visible to the 16-bit ISA (r2 to r7, r16, r17). In the 16-bit ISA, load and store offsets are shifted left until they are aligned to the data type being loaded or stored. This is done to provide a greater offset range. In the case of word accesses, the offset is left shifted by two bits. In the case of halfword accesses, the offset is left shifted by one bit.

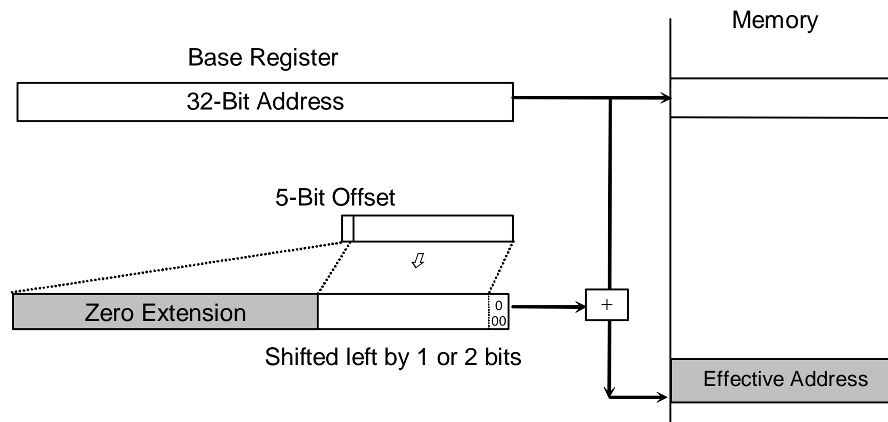


Figure 4-3 Register Indirect with Offset Addressing (16-Bit ISA)

SP-Relative with Offset Addressing

In the 32-bit ISA, there is no hardware-designated stack pointer. Although r29 is conventionally used to maintain the program stack pointer, any general-purpose register (except r0) can be used from the point of view of hardware. In the 16-bit ISA, however, one of the general-purpose registers, r29, serves as a stack pointer and is called sp. The 16-bit ISA references r29 implicitly through a special function code, thereby eliminating the base register field. This made it possible to expand the offset field to eight bits. The instruction format is the RI (register-immediate) type. In SP-relative addressing, the effective address is formed from a eight-bit offset (shifted left by two

bits) relative to the sp register. The LBU, LHU, LW, SB, SH and SW instructions can use this addressing mode. These instructions can address a range of 1 Kbytes (2^{10}) of memory without the need to EXTEND the instruction.

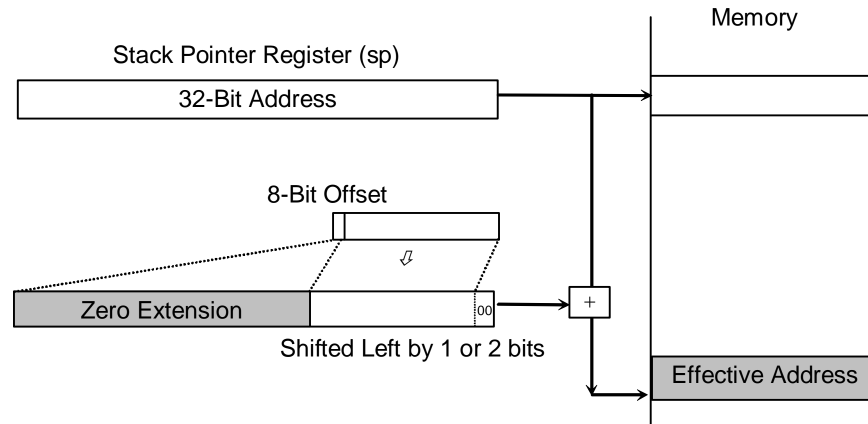


Figure 4-4 SP-Relative Addressing (16-Bit ISA))

FP-Relative with Offset Addressing

In the 16-bit ISA, r30 serves as a frame pointer (fp) register. The 16-bit ISA references r30 implicitly through a special function code, thereby eliminating the base register field. This made it possible to expand the offset field to five bits. The instruction format is the RI (register-immediate) type. For example, for 32-bit word access, the effective address is formed from a five-bit offset (shifted left by two bits) relative to the fp register with the zero extended value. The LBU, LHU, LW, SB, SH and SW instructions can use this addressing mode. These instructions can address a range of 128 bytes (27) of memory without the need to EXTEND the instruction.

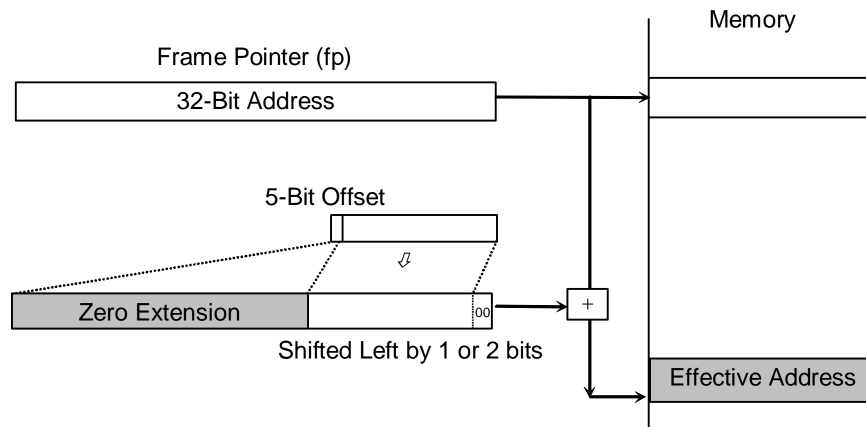


Figure 4-5 FP-Relative with Offset Addressing (16-Bit ISA, Word Access)

PC-Relative with Offset Addressing

PC-relative with offset addressing is supported by the Load Word (LW) instruction. In PC-relative with offset addressing, the effective address is formed by shifting the eight-bit offset left by two bits with the zero extended value and adding the resultant value to the PC with the lower two bits cleared. A 32-bit constant is then loaded into a register from the addressed memory location. 32-bit constants can be embedded in the code segment to get the maximum benefit from this addressing mode.

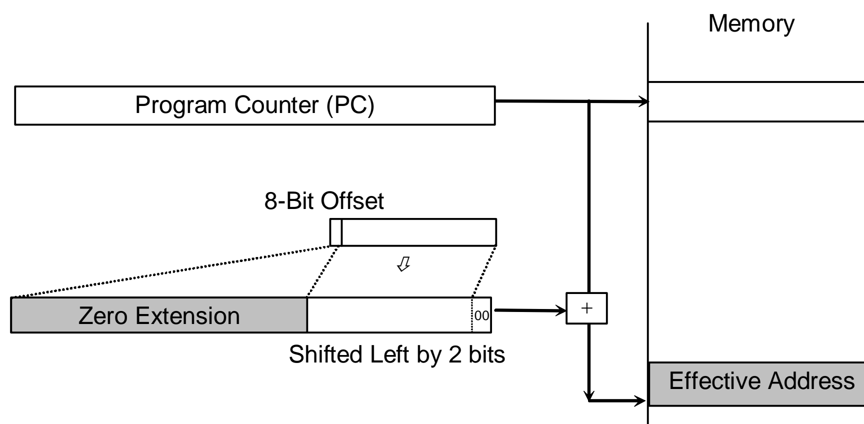


Figure 4-6 PC-Relative with Offset Addressing (16-Bit ISA)

4.2.2 Overview of Load and Store Instructions

Table 4-7 and Table 4-8 give the load and store instructions to perform byte, halfword and word accesses. The LB and LH instructions sign-extend the loaded byte and halfword respectively. The LBU and LHU instructions, which have the “U” (unsigned) suffix, zero-extend the loaded byte and halfword respectively and placed the result in the register.

Table 4-7 Load Instructions

Data Type	Unsigned Load	Signed Load	Addressing
Byte	LBU	LB	Register-Indirect, SP-Relative, FP-Relative
Halfword	LHU	LH	Register-Indirect, SP-Relative, FP-Relative
Word	LW	—	Register-Indirect, SP-Relative, FP-Relative, PC-Relative

Table 4-8 Store Instructions

Data Type	Opcode	Addressing
Byte	SB	Register-Indirect, SP-Relative, FP-Relative
Halfword	SH	Register-Indirect, SP-Relative, FP-Relative
Word	SW	Register-Indirect, SP-Relative, FP-Relative, PC-Relative

4.2.3 32-Bit Address Generation

In 16-bit ISA mode, the offset field is restricted to only 5 to 8 bits. However, EXTENDing an instruction to 32 bits allows the same order of offset value magnitude as is available in the 32-bit ISA (-32768 to 32767). If the offset is outside this range, you must put it in a general register prior to the load or store instruction. Alternatively, for word loads, you can use PC-relative with offset addressing. Three examples are given below.

- Example 1: Base address + 32-bit offset

In the example below, the ADDU (Add Unsigned) instruction is used to add the offset held in register r5 to the base address in register r4. The result is placed back into r4. Then the LW instruction uses r4 as the base register to address a memory location.

```
ADDU r4,r4,r5
LW r6,0(r4)
```

- Example 2: Base address + 32-bit offset

For offsets greater than 16 bits, the 32-bit ISA uses the LUI (Load Upper Immediate) instruction to load the upper 16 bits of a register, followed by a concatenation with the lower 16 bits using a logical OR instruction. Since the previous TX19 does not have the LUI instruction, a 32-bit offset is embedded in code and loaded from memory using PC-relative addressing. On the other hand, the TX19A now provides the LUI and ORI instructions, enabling the same coding as for the 32-bit ISA.

– TX19: Code efficient	– TX19A
LW r5,16(pc)	LUI r5,0x0008
ADDU r4,r4,r5	ORI r5,0x0234
LW r6,0(r4)	ADDU r4,r4,r5
	LW r6,0(r4)

- Example 3: Arbitrary 32-bit absolute address

In the example below, the first LW instruction loads a 32-bit absolute address from memory using PC-relative addressing. Then the second LW instruction can address a desired memory location, with an offset of zero.

```
LW r4,16(pc)
LW r6,0(r4)
```

The LUI and ORI instructions can also be used in combination to form a 32-bit absolute address:

```
LUI r4,0x0008
ORI r4,0x0234
```

4.2.4 SYNC Instruction

The memory synchronization instruction, SYNC, guarantees the sequence of memory references by interlocking the instruction pipeline until loads, stores and instruction fetches which performed prior to the present instruction are completed before loads or stores after this instruction are allowed to start.

4.3 Computational Instructions

This section describes the computational instructions available in the 16-bit ISA. Section 4.3.1 provides a category of computational instructions and an overview of the newly added instructions. Section 4.3.2 discusses computations that involve the use of 32-bit constants. For 64-bit arithmetic and rotate operations, see Chapter 3, *32-Bit ISA Summary and Programming Tips*, since the same instructions can be used to implement them in both the 32-bit and 16-bit ISA modes.

4.3.1 Overview of Computational Instructions

Computational instructions in the 16-bit ISA are categorized into four groups shown in Table 4-9. They consist of arithmetic, compare, logical, shift, multiply, divide and multiply-and-add instructions. Multiply-and-subtract instructions are not available in the 16-bit ISA. The 16-bit ISA does not support MIPS16 instructions for 64-bit, doubleword arithmetic and shift operations.

Table 4-9 Computational Instructions

Category	Instruction	Opcode
ALU Immediate	Add	ADDIU
	Set On Less Than	SLTI・SLTIU
	Compare	CMPI
	Load Immediate	LI・LUI
	Logical AND	ANDI
	Logical OR	ORI
	Logical XOR	XORI
2-and 3-Operand Register Type	Add	ADDU
	Subtract	SUBU
	Saturate	SADD・SSUB
	Set On Less Than	SLT・SLTU
	Compare	CMP
	Negate	NEG
	Logical AND	AND
	Logical OR	OR
	Logical XOR	XOR
	NOT	NOT
	MOVE	MOVE
	Bit Search	BS1F
	Bit Field	BFINS
	MAX/MIN	MAX・MIN
	Sign- and Zero-Extend	SEB・SEH・ZEB・ZEH
Shift	Logical Shift	SLL・SLLV・SRL・SRLV
	Arithmetic Shift	SRA・SRAV
Multiply and Divide	Multiply and Multiply-and-Add	MULT・MULTU・MADD・MADDU
	Divide	DIV・DIVU・DIVE・DIVEU
	Move From/To HI/LO	MFHI・MFLO・MTHI・MTLO

In ALU immediate instructions, the source operands are a general-purpose register and a 4- or 8-bit immediate. The new instructions, ANDI, ORI, XORI and LUI, are 32 bits in length, prepended with the EXTEND code. There are no 16-bit codes for these instructions; as such, they have a 16-bit immediate that is zero-extended and treated as a 32-bit unsigned operand (except the LUI instruction). Except for the ADDIU and LUI instructions, the 8-bit immediate in ALU immediate instructions are zero-extended. However, when EXTENDed, the immediate in the ADDIU, SLTI and SLTIU instructions are treated as a 16-bit signed integer (in the same manner as for the 32-bit ISA), and the immediate in other instructions are treated as a 16-bit unsigned integer.

Register-type instructions manipulate the values held in two general-purpose registers and place the result into a general-purpose register. The 16-bit ISA provides the CMP, NEG and NOT instructions. CMP compares the values in two registers. NEG performs two's complement of a value in a register. The NOT instruction performs one's complement of a value in a register. Additionally, the 16-bit ISA has the MOVE instruction to copy values between the eight registers plus the fp register and the remaining 24 registers of the full 32-bit architecture.
The 16-bit ISA has the Compare (CMP), Negate (NEG) and Not (NOT) instructions since these operations can not be synthesized from other instructions using r0 as a source. Compare instructions (CMP, CMPI) and set-on-less-than instructions (SLTI, SLTIU, SLT, SLTU) implicitly use register t8 (r24) as the destination.

The 16-bit ISA provides the same set of shift instructions as the 32-bit ISA. In the previous TX19, the *sa* field is only 3-bits wide; thus the shift amount is restricted to 1 to 8 (000 is defined as a shift of 8 bits). EXTEND enlarges the 3-bit *sa* field into 5 bits for a shift of 0 to 31 as in the 32-bit ISA. Additionally, the TX19A has also new instructions with a 5-bit *sa* field for a shift of 1 to 31 bits (the *sa* value of 00000 is undefined).

Multiply, divide and multiply-and-add instructions in the 16-bit ISA perform the same functionality as those in the 32-bit ISA. The multiply and multiply-and-add instructions in the 16-bit ISA can place the lower 32 bits of the result into a general-purpose register. The 16-bit ISA also provides the MTHI, MTLO, MFHI and MFLO instructions to access the HI and LO registers.

The TX19A offers new divide instructions (DIVE and DIVEU). The signed divide instruction (DIVE) generates an Integer Overflow exception when divide-by-zero or overflow conditions are detected, whereas the unsigned divide instruction (DIVEU) generates an Integer Overflow exception when a divide-by-zero condition is detected.

The TX19A provides the ZEB, ZEH, SEB and SHE instructions, new instructions that zero-extend or sign-extend a byte or halfword into 32 bits.

Additionally, the TX19A has saturate instructions (SADD and SSUB). For example, the SADD instruction adds the contents of general-purpose registers *rx* and *ry*; saturation clamps results to the largest representable positive number (0x7FFF_FFFF) on overflow and to the smallest representable negative number (0x8000_0000) on underflow. If overflow or underflow does not occur, the sum of *rx* and *ry* is placed into *ry*.

The new instructions MIN and MAX perform an arithmetic comparison on a pair of registers (*rx* and *ry*). The MIN instruction, for example, places the value of register *rx* into register *rz* if *rx* is less than *ry*, and otherwise, the value of *ry* into *rz*.

The bit field instruction (BFINS) helps the C compiler improve code density. C programs often deal with bit fields; the BFINS instruction copies a bit field from one register into another register with a single instruction. Also, the bit search instruction (BS1F) is convenient for scanning through an operand for a set bit, for example, for the purpose of key scanning in embedded control systems.

4.3.2 32-Bit Constants

With the previous TX19, even EXTEND can enlarge immediate fields in computational instructions to only 16 bits.

Since the 16-bit ISA of the TX19A has the LUI and ORI instructions, you can deal with 32-bit constants in the same manner as for the 32-bit ISA. Following is an example of adding a 32-bit constant to the contents of a general-purpose register:

```
LUI r5,0x8000
ORI r5,0x1234
ADDU r6,r6,r5
```

For code density, 32-bit constants can be embedded in the code segment, typically between subroutine bodies, in the previous TX19 way. Then the LW instruction can reference those 32-bit constants using PC-relative addressing. Even with the overhead of the constant storage, this is more compact than using a pair of the LUI and ORI instructions.

In the following example, the LW instruction loads a 32-bit constant into register r5 from memory. Then, the ADDU instruction adds the contents of r4 and r5 together and puts the results in r6.

```
LW r5,offset(pc)
ADDU r6,r4,r5
```

Zero Value

Generally, the 16-bit ISA does not have direct access to r0. When a value of zero is necessary, use the following LI (Load Immediate) instruction which zero-extends and loads the immediate value (0) into *rx*.

```
LI rx,0
```

Alternatively, you can use the MOVE instruction to get a value of zero. Since the MOVE instruction can move values between the eight registers visible to the 16-bit ISA and the remaining 24 registers of the full 32-bit architecture, the following gives you a value of zero:

```
MOVE ry,r0
```

4.4 Jump and Branch Instructions

This section describes the jump and branch instructions available in the 16-bit ISA, focusing on the differences from the 32-bit instructions. Section 4.4.1 gives an overview of jump and branch instructions. Section 4.4.2 provides programming tips for branching on arithmetic comparisons. Section 4.4.3 describes a technique to jump to 32-bit addresses.

4.4.1 Overview of Jump and Branch Instructions

The 16-bit ISA has no branch instruction that compares two registers and then branches, such as BEQ, BNE, BGEZ, BGTZ, BLEZ and BLTZ. To compensate for the loss of these instructions, the 16-bit ISA includes compare instructions (CMP, CMPI) to test if two registers or a register and an immediate are equal. Since these compare instructions and all set-on-less-than instructions set register t8, the 16-bit ISA provides branch instructions to test t8 and branch based on the zero or non-zero state of t8. The TX19A has a new branch-and-link instruction (BAL).
Even in 16-bit ISA mode, the JAL and JALX instructions are 32-bit wide to provide a large enough address field to jump to far procedures. Table 4-10 and Table 4-11 show the opcodes of the jump and branch instructions in the 16-bit ISA.

Table 4-10 Jump Instructions (16-Bit ISA)

Opcode	Name	Addressing
JAL	Jump And Link	Paged Absolute
JALX	Jump And Link Exchange	Paged Absolute
JR	Jump Register	Register Indirect
JRC	Jump Register, Compact	Register Indirect
JALR	Jump And Link Register	Register Indirect
JALRC	Jump And Link Register, Compact	Register Indirect

Table 4-11 Branch Instructions (16-Bit ISA)

Opcode	Name	Condition	Addressing
BEQZ	Branch On Equal to Zero	$rx = 0$	PC-relative
BNEZ	Branch On Not Equal Zero	$rx \neq 0$	PC-relative
BTEQZ	Branch On T8 Equal To Zero	$t8 > 0$	PC-relative
BTNEZ	Branch On T8 Not Equal To Zero	$t8 \neq 0$	PC-relative
B	Unconditional Branch	—	PC-relative
BAL	Branch And Link	—	PC-relative

Jump-and-link and BAL instructions save a return address in register r31. They are typically used for subroutine calls.

Branch instructions in the 16-bit ISA use the same addressing mode as those in the 32-bit ISA. However, since instructions are 16-bits wide, the branch address is shifted by one bit, not by two bits. The offset immediate is either 8-bits or 11-bits wide.

■ Delayed Branch

In the 16-bit ISA, there is no delayed branch. Branches always take effect before the next instruction. Therefore, there is no restriction on the instructions that follow a branch instruction. Instructions following a branch are executed only when the branch is not taken.

As in the 32-bit ISA mode, jump instructions in the 16-bit ISA have a delay slot, except the JRC and JALRC instructions, new compact versions of JR and JALR.

■ Run-Time Switching of the ISA Modes

As shown in Table 4-1, the 16-bit ISA includes the JALX, JR, JALR, JRC and JALRC instructions. These instructions can be used in 16-bit ISA mode to toggle the ISA mode bit in the PC and switch to the other ISA mode. See Section 3.4.3, *Run-Time Switching of the ISA Modes*, for details on this.

■ Subroutine Calls

The 16-bit ISA has jump-and-link instructions (JALX, JALR) and a branch-and-link instruction (BAL). See Section 3.4.7, *Subroutine Calls*, for details on subroutine calls.

4.4.2 Branching on Arithmetic Comparisons

As mentioned in the previous section, the 16-bit ISA did away with instructions that compare two registers and branch like "BEQ r10, r7, Equal". Also, set-on-less-than instructions (SLT, SLTU) in the 16-bit ISA are two-register instructions instead of three. In the 16-bit ISA, the SLT and SLTU instructions implicitly set register t8 based on the equality of the values in two registers. Because of this, the 16-bit ISA has new instructions, BTEQZ and BTNEZ, to test the t8 register to see if it is zero or not.

As explained in Section 3.4.5, *Branching on Arithmetic Comparisons*, in 32-bit ISA mode, ORI and BEQ (or BNE) are used in pair to compare the contents of a register and an immediate:

```
ORI r10,r0,0x1234
BEQ r10,r7,Label
```

Since the TX19A now has the LUI and ORI instructions, the 16-bit ISA routine can use the same sequence of instructions to branch on an arithmetic comparison. Also, the 16-bit ISA provides the CMPI instruction that compares a register and an immediate and sets t8 based on their equality.

The following gives three examples of compare and branch in 16-bit ISA mode.

- Example 1: Branch if r6 ≥ r7

The following sequence of instructions checks if the contents of r6 is equal to or greater than the contents of r7. If r6 is less than r7, the SLT (Set On Less Than) instruction sets t8 to one. Otherwise, t8 is set to zero. The BTEQZ instruction branches to Label if t8 is zero.

```
SLT r6,r7
BTEQZ Label
```

- Example 2: Branch if r7 ≥ 0x1234

The following sequence of instructions checks if the contents of r7 are equal to or greater than 0x1234. In this example, the SLTI (Set On Less Than Immediate) instruction implicitly sets t8 based on the magnitude of r7 and 0x1234. Then the BTEQZ instruction branches to Label if t8 is equal to zero.

```
SLTI r7,0x1234
BTEQZ Label
```

- Example 3: Branch if r7 = 0x1234

The following sequence of instructions checks the equality of the contents of a register and an immediate value. In this example, the CMPI (Compare Immediate) instruction compares the contents of r7 to 0x1234 and sets t8 to 0 if they are equal. (CMPI actually exclusive-ORs two values.)

```
CMPI r7,0x1234
BTEQZ Label
```

4.4.3 Jumping to 32-Bit Addresses

Since the 16-bit ISA of the TX19A has the LUI and ORI instructions, you can deal with 32-bit addresses in the same manner as for the 32-bit ISA. For code density, 32-bit constants can be embedded in the code segment, typically between subroutine bodies, in the previous TX19 way; then the LW instruction can reference those 32-bit constants using PC-relative addressing.

– For code density	– For execution speed
`LW r4,0(pc)`	`LUI r4,0x0008`
`JR r4`	`ORI r4,0x0234`
	`JR r4`

There is also an instruction (ADDIU, *rx*, pc, *immediate*) to calculate a PC-relative address and place it in a register.

4.5 Bit Manipulation Instructions

The TX19A provides bit manipulation instructions that test or modify a bit in memory. The previous TX19 not only requires multiple instructions to manipulate a bit in memory, but also may need additional instructions to disable interrupts during a bit manipulation operation. The TX19A performs memory bit manipulation with one instruction, helping to decrease code size and increase execution speed.

Table 4-12 Bit Manipulation Instructions

Opcode	Name	Destination
BTST	Bit Test	t8register
BEXT	Bit Extract	t8register
BCLR	Bit Clear	Memory
BSET	Bit Set	Memory
BINS	Bit Insert	Memory
ADDMIU	Add Immediate to Memory Word	Memory

4.6 SAVE and RESTORE Instructions

The TX19A provides the SAVE and RESTORE instructions for stack operations. The SAVE instruction saves a set of CPU registers to memory stack with one instruction. The RESTORE instruction restores a set of CPU registers from memory stack with one instruction. These instructions help to decrease code size, as compared to the TX19 that require multiple instructions for stack operations.

Table 4-13 SAVE and RESTORE Instructions

Opcode	Name	Registers Saved or Restored
SAVE	Save Registers and Set up Stack Frame	r4-r7, r16, r17, r18-r23, r30, r31
RESTORE	Restore Registers and Deal locate Stack Frame	r4-r7, r16, r17, r18-r23, r30, r31

4.7 System Control Coprocessor (CP0) Instructions

The previous TX19 does not have CP0 instructions; it requires that the ISA mode be switched to 32-bit ISA (with the JALX instruction, etc.) in order to access the system control coprocessor (CP0). The 16-bit ISA of the TX19A now provides CP0 instructions, with the restriction that the IER, Config1, Config2 and Config3 registers are inaccessible in 16-bit ISA mode.
For example, in 32-bit ISA mode, the Interrupt Enable (IE) bit in the Status register can be modified by writing a zero or non-zero value to the IER register. However, in 16-bit ISA mode, the CP0 instruction can not access the IER register, ; to compensate for this restriction, the 16-bit ISA provides the EI and DI instructions that sets or clears the IE bit.

Table 4-14 System Control Coprocessor (CP0) Instructions

Opcode	Name
MFC0	Move from Coprocessor 0
MTC0	Move to Coprocessor 0
AC0IU	Add Coprocessor 0 Immediate Unsigned.

4.8 Special Instructions

Special instructions include the BREAK (Breakpoint) and SDBBP (Software Debug Breakpoint) instructions as well as the new EI (Enable Interrupt), DI (Disable Interrupt), SYSCALL (System Call), SYNC (Synchronize), ERET (Exception Return), DERET (Debug Exception Return) and WAIT (Enter Standby Mode) instructions.

The 16-bit ISA provides an instruction called EXTEND. The purpose of the EXTEND instruction is twofold. First, the EXTEND instruction consists of a 5-bit opcode and an 11-bit immediate value. In this case, EXTEND does not generate a MIPS machine instruction on its own, but instead contributing the 11-bit immediate to be concatenated with the immediate data carried in the following 16-bit instruction. This way, EXTEND extends a 16-bit instruction to 32 bits, providing large immediate values, as shown in Table 4-9. Second, the 16-bit ISA has several 32-bit instructions prepended with the EXTEND code. In such instructions, the 11-bit immediate field is replaced with an opcode. The new SYNC, ERET, DERET, WAIT, BS1F, MAX and MIN instructions are EXTENDed instructions and have no 16-bit equivalents.

Table 4-15 EXTENDable Instructions

16-Bit Instruction		Immediate Bit Size	
		Before EXTENDed	After EXTENDed
Load/Store	LB・LBU	5 (or 7)	16
	LH・LHU	5 (or 6)	16
	LW	5 (or 8)	16
	SB	5 (or 7)	16
	SH	5 (or 6)	16
	SW	5 (or 8)	16
Computational	ADDIU	4	15
		8	16
	SLTI・SLTIU	8	16
	CMPI	8	16
	LI	8	16
	LUI	—	16
	SLL	3	5
	SRL	3	5
	SRA	3	5
	ANDI	—	16
	ORI	—	16
	XORI	—	16
	LUI	—	16
	ADDMIU	—	14

16-Bit Instruction		Immediate Bit Size	
		Before EXTENDed	After EXTENDed
Branch	BEQZ	8	16
	BNEZ	8	16
	BTEQZ	8	16
	BTNEZ	8	16
	B	11	16
	BAL	8	16
Bit Manipulation	BTST	5	14
	BEXT	5	14
	BCLR	5	14
	BSET	5	14
	BINS	5	14
SAVE・RESTORE	SAVE	4	8
	RESTORE	4	8
Bit Field	BFINS	—	5

EXTEND does not need to start on a word boundary. There is one restriction on the use of EXTEND; it may not be placed in a jump delay slot since its outcome is undefined. You do not need to explicitly place EXTEND before a 16-bit instruction with an immediate field. If you specify an immediate longer than permitted in the 16-bit ISA, the assembler will automatically break it down to smaller immediates using EXTEND. For example, the instruction:

```
ADDIU r3,0x1234
```

is an RI (register-immediate) type instruction, and the immediate field is only 8-bits long. Thus, this instruction is EXTENDed to 32 bits using the EXT-RI instruction format. This is illustrated in Figure 4-16.

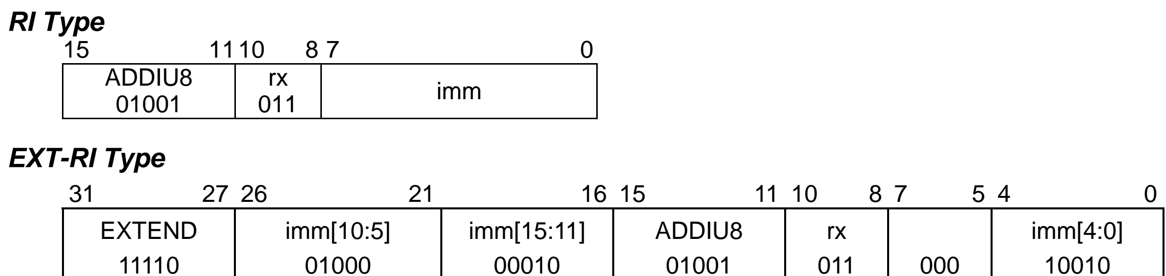


Figure 4-16 RI Format vs. EXT-RI Format

"ADDIU, *ry*, *rx*, *immediate*" has a 4-bit immediate field. Since EXTEND can only supply 11 more bits, it enlarges the immediate only to 15 bits.

Also, even when EXTENDed, the SLL, SRL and SRA instructions have a 5-bit immediate; the bit manipulation instructions have a 14-bit immediate; and the BFINS instruction has a 5-bit immediate (see Table 4-15).

4.9 Instruction Summary

This section provides an overview of the instructions in the 16-bit ISA.

■ Notational Conventions

In this section, all variable fields in an instruction format are shown in italicized lowercase letters, like *rx*, *ry*, *rz*, *immediate* and *sa* (shift amount). For the sake of clarity, an alias is sometimes used to refer to a field in the formats of specific instructions. For example, *base* and *offset* are used instead of *rx* and *immediate* in the formats of load and store instructions. Certain instructions can use r24 (t8), r28 (gp), r29 (sp), r30 (fp) and r31 (ra) for specific purposes. These registers are shown as t8, gp, sp, fp and ra. HI and LO are the special registers that hold the results of integer multiply and divide operations.

■ Instructions Not Implemented in the TX19A

The TX19A does not provide support for the MIPS16eASE instructions that manipulate 64-bit doubleword operands. See Appendix D for a list of complete comparisons among the TX19A, the TX19 and the MIPS16.

Table 4-17 Load and Store Instructions (16-Bit ISA)

Instruction	Format	Operation
Load Byte	LB *ry*, *offset*(*base*)	The 5-bit *offset* is zero-extended and added to *base* to form an effective address. The byte in memory addressed by the EA is sign-extended and loaded into *ry*.
Load Byte Unsigned	LBU *ry*, *offset*(*base*)	The 5-bit *offset* is zero-extended and added to *base* to form an effective address. The byte in memory addressed by the EA is zero-extended and loaded into *ry*.
	* LBU *ry*, *offset*(*sp*)	The 7-bit *offset* is zero-extended and added to sp to form an effective address. The byte in memory addressed by the EA is zero-extended and loaded into *ry*.
	* LBU *ry*, *offset*(*fp*)	The 7-bit *offset* is zero-extended and added to fp to form an effective address. The byte in memory addressed by the EA is zero-extended and loaded into *ry*.
Load Halfword	LH *ry*, *offset*(*base*)	The 5-bit *offset* is shifted left by one bit, zero-extended and added to *base* to form an effective address. The halfword in memory addressed by the EA is sign-extended and loaded into *ry*.
Load Halfword Unsigned	LHU *ry*, *offset*(*base*)	The 5-bit *offset* is shifted left by one bit, zero-extended and added to *base* to form an effective address. The halfword in memory addressed by the EA is zero-extended and loaded into *ry*.
	* LHU *ry*, *offset*(*sp*)	The 6-bit *offset* is shifted left by one bit, zero-extended and added to sp to form an effective address. The halfword in memory addressed by the EA is zero-extended and loaded into *ry*.
	* LHU *ry*, *offset*(*fp*)	The 6-bit *offset* is shifted left by one bit, zero-extended and added to fp to form an effective address. The halfword in memory addressed by the EA is zero-extended and loaded into *ry*.

* Enhancements from the TX19 to the TX19A

Instruction	Format	Operation
Load Word	LW *ry*, *offset*(*base*)	The 5-bit *offset* is shifted left by two bits, zero-extended and added to *base* to form an effective address. The word in memory addressed by the EA is loaded into *ry*.
	LW *rx*, *offset(pc)*	The 8-bit *offset* is shifted left by two bits, zero-extended and added to the masked PC value (i.e., PC value with the lower two bits cleared) to form an effective address. The word in memory addressed by the EA is loaded into *rx*.
	LW *rx*, *offset(sp)*	The 8-bit *offset* is shifted left by two bits, zero-extended and added to sp to form an effective address. The word in memory addressed by the EA is loaded into *rx*.
	* LW *ry*, *offset(fp)*	The 5-bit *offset* is shifted left by two bits, zero-extended and added to fp to form an effective address. The word in memory addressed by the EA is loaded into *ry*.
Store Byte	SB *ry*, *offset*(*base*)	The 5-bit *offset* is zero-extended and added to *base* to form an effective address. The least-significant byte in *ry* is stored in memory addressed by the EA.
	* SB *ry*, *offset*(*sp*)	The 7-bit *offset* is zero-extended and added to sp to form an effective address. The least-significant byte in *ry* is stored in memory addressed by the EA.
	* SB *ry*, *offset*(*fp*)	The 7-bit *offset* is zero-extended and added to fp to form an effective address. The least-significant byte in *ry* is stored in memory addressed by the EA.
Store Halfword	SH *ry*, *offset*(*base*)	The 5-bit *offset* is shifted left by one bit, zero-extended and added to *base* to form an effective address. The low-order halfword in *ry* is stored in memory addressed by the EA.
	* SH *ry*, *offset*(*sp*)	The 6-bit *offset* is shifted left by one bit, zero-extended and added to sp to form an effective address. The low-order halfword in *ry* is stored in memory addressed by the EA.
	* SH *ry*, *offset*(*fp*)	The 6-bit *offset* is shifted left by one bit, zero-extended and added to fp to form an effective address. The low-order halfword in *ry* is stored in memory addressed by the EA.
Store Word	SW *ry*, *offset*(*base*)	The 5-bit *offset* is shifted left by two bits, zero-extended and added to *base* to form an effective address. *ry* is stored in memory addressed by the EA.
	SW *rx*, *offset(sp)*	The 8-bit *offset* is shifted left by two bits, zero-extended and added to sp to form an effective address. *rx* is stored in memory addressed by the EA.
	SW ra, *offset(sp)*	The 8-bit *offset* is shifted left by two bits, zero-extended and added to sp to form an effective address. ra is stored in memory addressed by the EA.
	* SW ry, *offset(fp)*	The 5-bit *offset* is shifted left by two bits, zero-extended and added to fp to form an effective address. *ry* is stored in memory addressed by the EA.

* Enhancements from the TX19 to the TX19A

Table 4-18 ALU Immediate Instructions (16-Bit ISA)

Instruction	Format	Operation
Add Immediate	ADDIU *ry*, *rx*, *immediate*	The 4-bit *immediate* is sign-extended and added to *rx*. The result is placed into *ry*. Does not cause an exception on 2's-complement overflow.
	ADDIU *rx*, *immediate*	The 8-bit *immediate* is sign-extended and added to *rx*. The result is placed back into *rx*. Does not cause an exception on 2's-complement overflow.
	ADDIU sp, *immediate*	The 8-bit *immediate* is shifted left by three bits and sign-extended. The resultant value is added to sp and the sum is placed back into sp. Does not cause an exception on 2's-complement overflow.
	* ADDIU fp, *immediate*	The 8-bit *immediate* is shifted left by two bits and sign-extended. The resultant value is added to fp and the sum is placed back into fp. Does not cause an exception on 2's-complement overflow.
	ADDIU *rx*, pc, *immediate*	The 8-bit *immediate* is shifted left by two bits and zero-extended. The resultant value is added to the masked PC value (i.e., PC value with the lower two bits cleared) and the sum is placed into *rx*. Does not cause an exception on 2's-complement overflow.
	ADDIU *rx*, sp, *immediate*	The 8-bit *immediate* is shifted left by two bits and zero-extended. The resultant value is added to sp and the sum is placed into *rx*. Does not cause an exception on 2's-complement overflow.
Set On Less Than Immediate	SLTI *rx*, *immediate*	t8 = 1 if *rx* is less than *immediate*; otherwise t8 = 0. The 8-bit *immediate* is zero-extended. Two values are compared as signed integers.
Set On Less Than Immediate Unsigned	SLTIU *rx*, *immediate*	t8 = 1 if *rx* is less than *immediate*; otherwise t8 = 0. The 8-bit *immediate* is zero-extended. Two values are compared as unsigned integers.
Compare Immediate	CMPI *rx*, *immediate*	t8 = 0 if *rx* = *immediate*; otherwise t8 ≠ 0. The 8-bit *immediate* is zero-extended.
Load Immediate	LI *rx*, *immediate*	The 8-bit immediate is zero-extended and loaded into *rx*.
* Logical AND Immediate	ANDI *ry*, *immediate*	The contents of *ry* is ANDed with *immediate* and the result is placed back into *ry*. The 16-bit *immediate* is zero-extended.
* Logical OR Immediate	ORI *ry*, *immediate*	The contents of *ry* is ORed with *immediate* and the result is placed back into *ry*. The 16-bit *immediate* is zero-extended.
* Logical Exclusive_OR Immediate	XORI *ry*, *immediate*	The contents of *ry* is exclusive-ORed with *immediate* and the result is placed back into *ry*. The 16-bit *immediate* is zero-extended.
* Load Upper Immediate	LUI *ry*, *immediate*	The 16-bit *immediate* is shifted left by 16 bits and concatenated to 16 bits of zeros. The result is placed into *ry*.

* Enhancements from the TX19 to the TX19A

Table 4-19 Register-Type Instructions (16-Bit ISA)

Instruction	Format	Operation
Add Unsigned	ADDU *rz*, *rx*, *ry*	The sum *rx* + *ry* is placed into *rz*. Does not cause an exception on 2's-complement overflow.
Subtract Unsigned	SUBU *rz*, *rx*, *ry*	The remainder *rx* - *ry* is placed into *rz*. Does not cause an exception on 2'scomplement overflow.
Set On Less Than	SLT *rx*, *ry*	t8 = 1 if *rx* is less than *ry*; otherwise t8 = 0. Two values are compared as signed integers.
Set On Less Than Unsigned	SLTU *rx*, *ry*	t8 = 1 if *rx* is less than *ry*; otherwise t8 = 0. Two values are compared as unsigned integers.
Compare	CMP *rx*, *ry*	t8 = 0 if *rx* is equal to *ry*; otherwise t8 = 0.
Negate	NEG *rx*, *ry*	*rx* = 0 - *ry* (2's-complement)
AND	AND *rx*, *ry*	The contents of *rx* is ANDed with the contents of *ry* and the result is placed back into *rx*.
OR	OR *rx*, *ry*	The contents of *rx* is ORed with the contents of *ry* and the result is placed back into *rx*.
Exclusive-OR	XOR *rx*, *ry*	The contents of *rx* is exclusive-ORed with the contents of *ry* and the result is placed back into *rx*.
Not	NOT *rx*, *ry*	*ry* is inverted bitwise and the result is placed into *rx*. (1'scomplement)
Move	MOVE *ry, r32*	The contents of *r32* is copied into *ry*.
	MOVE *r32, rz*	The contents of *rz* are copied into *r32*.
	* MOVE *fp, r32*	The contents of *r32* is copied into fp.
* Bit Search One Forward	BS1F *ry, rx*	*rx* is searched for the first set bit, starting from bit 0 towards bit 31. If a set bit is found in *rx*, its bit position (bit number plus 1) is placed into *ry*. If no set bit is found in *rx*, the value written to *ry* is 0.
* Bit Field Insert	BFINS *ry, rx, bit2, bit1*	A bit field indicated by [(*bit2* – *bit1*):0] in *rx* is copied into a location indicated by (*bit2*:*bit1*) in *ry*.
* Maximum Signed	MAX *rz, rx, ry*	The contents of *rx* is compared to the contents of *ry* as signed values. If *rx* is greater than *ry*, the value of *rx* is written to *rz*. Otherwise, the value of *ry* is written to *rz*.
* Minimum Signed	MIN *rz, rx, ry*	The contents of *rx* is compared to the contents of *ry* as signed values. If *rx* is less than *ry*, the value of *rx* is written to *rz*. Otherwise, the value of *ry* is written to *rz*.
* Sign-Extend Byte	SEB *rx*	The least-significant byte in *rx* is sign-extended. The result is placed back into *rx*.
* Sign-Extend Halfword	SEH *rx*	The low-order halfword in *rx* is sign-extended. The result is placed back into *rx*.
* Zero-Extend Byte	ZEB *rx*	The least-significant byte in *rx* is zero-extended. The result is placed back into *rx*.
* Zero-Extend Halfword	ZEH *rx*	The low-order halfword in *rx* is zero-extended. The result is placed back into *rx*.
* Saturated Additional	SADD *ry, rx, ry*	The contents of *rx* are added to the contents of *ry*. The sum saturates to the largest representable positive number (0x7FFF_FFFF) on overflow and to the smallest representable negative number (0x8000_0000) on underflow. The result is placed into *ry*. If overflow or underflow does not occur, the sum of *rx* and *ry* is placed into *ry*.
* Saturated Subtraction	SSUB *ry, rx, ry*	The contents of *ry* are subtracted from the contents of *rx*. On overflow, the remainder saturates to the largest representable positive number (0x7FFF_FFFF) if *rx* is zero or a positive number and to the smallest representable negative number (0x8000_0000) if *rx* is a negative number. The result is placed into *ry*. If overflow does not occur, the remainder is placed into *ry*.

* Enhancements from the TX19 to the TX19A

Table 4-20 Shift Instructions (16-Bit ISA)

Instruction	Format	Operation
Shift Left Logical	SLL *rx*, *ry*, *sa*	The contents of *ry* are shifted left by *sa* bits. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into *rx*.
	* SLL *ry*, *sa*	The contents of *ry* are shifted left by *sa* bits. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed back into *ry*.
Shift Left Logical Variable	SLLV *ry*, *rx*	The contents of *ry* is shifted left the number of bits specified by the five least-significant bits of *rx*. Zeros are supplied to the vacated positions on the right.
Shift Right Logical	SRL *rx*, *ry*, *sa*	The contents of *ry* are shifted right by *sa* bits. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into *rx*.
	* SRL *ry*, *sa*	The contents of *ry* are shifted right by *sa* bits. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed back into *ry*.
Shift Right Logical Variable	SRLV *ry*, *rx*	The contents of *ry* is shifted right the number of bits specified by the five least-significant bits of *rx*. The 32-bit result is placed back into *ry*.
Shift Right Arithmetic	SRA *rx*, *ry*, *sa*	The contents of *ry* are shifted right by *sa* bits. The sign bit is copied to the vacated positions on the left. The 32-bit result is placed into *rx*.
	* SRA *ry*, *sa*	The contents of *ry* are shifted right by *sa* bits. The sign bit is copied to the vacated positions on the left. The 32-bit result is placed back into *ry*.
Shift Right Arithmetic Variable	SRAV *ry*, *rx*	The contents of *ry* is shifted right the number of bits specified by the five least-significant bits of *rx*. The sign bit is copied to the vacated positions on the left.

* Enhancements from the TX19 to the TX19A

Table 4-21 SAVE and RESTORE Instructions (16-Bit ISA)

Instruction	Format	Operation
* SAVE	SAVE *reg_list3, framesize4*	A set of registers indicated by *reg_list3* is saved to memory, and the contents of sp are adjusted by *framesize4*.
	SAVE *reg_list3, xsregs, aregs, framesize8*	A set of registers indicated by *reg_list3*, *xsregs* and *aregs* is saved to memory, and the contents of sp are adjusted by *framesize8*.
* RESTORE	RESTORE *reg_list3, framesize4*	A set of registers indicated by *reg_list3* is restored from memory, and the contents of sp are adjusted by *framesize4*.
	RESTORE *reg_list3, xsregs, aregs, framesize8*	A set of registers indicated by *reg_list3*, *xsregs* and *aregs* is restored from memory, and the contents of sp is adjusted by *framesize8*.

* Enhancements from the TX19 to the TX19A

Table 4-22 Multiply and Divide Instructions (16-Bit ISA)

Instruction	Format	Operation
Multiply	MULT *rx*, *ry*	The multiplicand is the signed value of *rx*. The multiplier is the signed value of *ry*. The 64-bit product *rx* * *ry* is placed into registers HI and LO.
	* MULT *ry, rx, ry*	The multiplicand is the signed value of *rx*. The multiplier is the signed value of *ry*. The 64-bit product *rx* * *ry* is placed into registers HI and LO. The low-order 32 bits of the product are copied into *ry*.
Multiply Unsigned	MULTU *rx*, *ry*	The multiplicand is the unsigned value of *rx*. The multiplier is the unsigned value of *ry*. The 64-bit product *rx* * *ry* is placed into registers HI and LO.
	* MULTU *ry, rx, ry*	The multiplicand is the unsigned value of *rx*. The multiplier is the unsigned value of *ry*. The 64-bit product *rx* * *ry* is placed into registers HI and LO. The low-order 32 bits of the product are copied into *ry*.
* Multiply And Add	MADD *rx*, *ry*	The multiplicand is the signed value of *rx*. The multiplier is the signed value of *ry*. The 64-bit product *rx* * *ry* is added to the contents of registers HI and LO and the result is placed back into HI and LO.
* Multiply And Add Unsigned	MADDU *rx*, *ry*	The multiplicand is the unsigned value of *rx*. The multiplier is the unsigned value of *ry*. The 64-bit product *rx* * *ry* is added to the contents of registers HI and LO and the result is placed back into HI and LO.
Divide	DIV *rx*, *ry*	The dividend is the signed value of *rx*. The divisor is the signed value of *ry*. The quotient is placed into register LO and the remainder is placed into register HI.
Divide Unsigned	DIVU *rx*, *ry*	The dividend is the unsigned value of *rx*. The divisor is the unsigned value of *ry*. The quotient is placed into register LO and the remainder is placed into register HI.
* Divide Exception	DIVE *rx*, *ry*	The dividend is the signed value of *rx*. The divisor is the signed value of *ry*. The quotient is placed into register LO and the remainder is placed into register HI. An Integer Overflow exception occurs if divide-by-zero or overflow conditions are detected.
* Divide Exception Unsigned	DIVEU *rx*, *ry*	The dividend is the unsigned value of *rx*. The divisor is the unsigned value of *ry*. The quotient is placed into register LO and the remainder is placed into register HI. An Integer Overflow exception occurs if a divide-by-zero condition is detected.
Move From HI	MFHI *rx*	The contents of register HI is copied to *rx*.
Move From LO	MFLO *rx*	The contents of register LO are copied to *rx*.
* Move to HI	MTHI *rx*	The contents of *rx* are copied to register HI.
* Move to LO	MTLO *rx*	The contents of *rx* are copied to register LO.

* Enhancements from the TX19 to the TX19A

Table 4-23 Bit Manipulation Instructions (16-Bit ISA)

Instruction	Format		Operation
* Bit Test	BTST	*offset(base3), pos3*	A bit specified by *pos3* in a memory byte is negated and placed into the least-significant bit (LSB) of t8. The upper 31 bits of t8 are filled with zeros. The effective address is computed by zero-extending the 14-bit *offset* and adding the resultant value to the contents of *base3*.
	BTST	*offset(*r0*), pos3*	A bit specified by *pos3* in a memory byte is negated and placed into the least-significant bit (LSB) of t8. The upper 31 bits of t8 are filled with zeros. The effective address is computed by sign-extending the 14-bit *offset* and adding the resultant value to the contents of r0.
	BTST	*offset(fp), pos3*	A bit specified by *pos3* in a memory byte is negated and placed into the least-significant bit (LSB) of t8. The upper 31 bits of t8 are filled with zeros. The effective address is computed by zero-extending the 5-bit *offset* and adding the resultant value to the contents of fp.
* Bit Extract	BEXT	*offset(base3), pos3*	A bit specified by *pos3* in a memory byte is copied into the least-significant bit (LSB) of t8. The upper 31 bits of t8 are filled with zeros. The effective address is computed by zero-extending the 14-bit *offset* and adding the resultant value to the contents of *base3*.
	BEXT	*offset(*r0*), pos3*	A bit specified by *pos3* in a memory byte is copied into the least-significant bit (LSB) of t8. The upper 31 bits of t8 are filled with zeros. The effective address is computed by sign-extending the 14-bit *offset* and adding the resultant value to the contents of r0.
	BEXT	*offset(fp), pos3*	A bit specified by *pos3* in a memory byte is copied into the least-significant bit (LSB) of t8. The upper 31 bits of t8 are filled with zeros. The effective address is computed by zero-extending the 5-bit *offset* and adding the resultant value to the contents of fp.
* Bit Clear	BCLR	*offset(base3), pos3*	A bit specified by *pos3* in a memory byte is cleared. The effective address is computed by zero-extending the 14-bit *offset* and adding the resultant value to the contents of *base3*.
	BCLR	*offset(*r0*), pos3*	A bit specified by *pos3* in a memory byte is cleared. The effective address is computed by sign-extending the 14-bit *offset* and adding the resultant value to the contents of r0.
	BCLR	*offset(fp), pos3*	A bit specified by *pos3* in a memory byte is cleared. The effective address is computed by zero-extending the 5-bit *offset* and adding the resultant value to the contents of fp.
* Bit Set	BSET	*offset(base3), pos3*	A bit specified by *pos3* in a memory byte is set. The effective address is computed by zero-extending the 14-bit *offset* and adding the resultant value to the contents of *base3*.
	BSET	*offset(*r0*), pos3*	A bit specified by *pos3* in a memory byte is set. The effective address is computed by sign-extending the 14-bit *offset* and adding the resultant value to the contents of r0.
	BSET	*offset(fp), pos3*	A bit specified by *pos3* in a memory byte is set. The effective address is computed by zero-extending the 5-bit *offset* and adding the resultant value to the contents of fp.

Instruction	Format		Operation
* Bit Insert	BINS	*offset(base3), pos3*	The least-significant bit (LSB) of t8 is copied into a bit position indicated by *pos3* in a memory byte. The effective address is computed by zero-extending the 14-bit *offset* and adding the resultant value to the contents of *base3*.
	BINS	*offset(*r0*), pos3*	The least-significant bit (LSB) of t8 is copied into a bit position indicated by *pos3* in a memory byte. The effective address is computed by sign-extending the 14-bit *offset* and adding the resultant value to the contents of r0.
	BINS	*offset(fp), pos3*	The least-significant bit (LSB) of t8 is copied into a bit position indicated by *pos3* in a memory byte. The effective address is computed by zero-extending the 5-bit *offset* and adding the resultant value to the contents of fp.
* Add Immediate to Memory Word	ADDMIU	*offset(base3), imm*	The 14-bit *offset* is shifted left by two bits, zero-extended, then added to the contents of *base3* to form an effective address (EA). The value indicated by *imm* is added to the memory word addressed by the EA, and the sum is written back to the EA.
	ADDMIU	*offset(*r0*), imm*	The 14-bit *offset* is shifted left by two bits, sign-extended, then added to the contents of r0 to form an effective address (EA). The value indicated by *imm* is added to the memory word addressed by the EA, and the sum is written back to the EA.

* Enhancements from the TX19 to the TX19A

Table 4-24 System Control Coprocessor (CP0) Instructions (16-Bit ISA)

Instruction	Format		Operation
* Move To Coprocessor 0	MTC0	*rx, cp0rd32*	The contents of *rx* are loaded into CP0 register *cp0rd32*.
* Move From Coprocessor 0	MFC0	*ry, cp0rs32*	The contents of CP0 register *cp0rs32* is loaded into *ry*.
* Add Coprocessor 0 Immediate Unsigned	AC0IU	*cp0rt32, immediate*	The value indicated by *immediate* is added to the contents of CP0 register *cp0rt32*. The result is placed back into *cp0rt32*.

* Enhancements from the TX19 to the TX19A

Table 4-25 Jump Instructions (16-Bit ISA)

Instruction	Format	Operation
Jump And Link	JAL *target*	The program jumps to the address computed using paged absolute addressing, i.e., by shifting the 26-bit *target* left by two bits and combining it with the four most-significant bits of PC + 4. The address of the instruction following the delay slot is saved in r31.
Jump And Link exchange	JALX *target*	The program jumps to the address using paged absolute addressing, i.e., by shifting the 26-bit *target* left by two bits and combining it with the four most-significant bits of PC + 4. The address of the instruction following the delay slot is saved in r31. The ISA mode bit in the PC toggles.
Jump Register	JR *rx*	The program jumps to the address specified by the upper 31 bits of *rx*. The least-significant bit of *rx* is interpreted as the ISA mode specifier.
	JR ra	The program jumps to the address specified by the upper 31 bits of ra. The least-significant bit of ra is interpreted as the ISA mode specifier.
* Jump Register, Compact	JRC *rx*	The program jumps to the address specified by the upper 31 bits of *rx*. The least-significant bit of *rx* is interpreted as the ISA mode specifier. This instruction does not have a delay slot.
	JRC ra	The program jumps to the address specified by the upper 31 bits of ra. The least-significant bit of ra is interpreted as the ISA mode specifier. This instruction does not have a delay slot.
Jump And Link Register	JALR ra, *rx*	The program jumps to the address specified by the upper 31 bits of *rx*. The least-significant bit of *rx* is interpreted as the ISA mode specifier. The address of the instruction following the delay slot is saved in ra.
* Jump And Link Register, Compact	JALRC ra, *rx*	The program jumps to the address specified by the upper 31 bits of *rx*. The least-significant bit of *rx* is interpreted as the ISA mode specifier. The address of the instruction following the delay slot is saved in ra. This instruction does not have a delay slot.

* Enhancements from the TX19 to the TX19A

Table 4-26 Branch Instructions (16-Bit ISA)

Instruction	Format	Operation
Branch On Equal To Zero	BEQZ *rx*, *offset*	If *rx* = 0, the program branches to the target address specified as a 8-bit *offset* relative to PC + 2 (or PC + 4 when EXTENDed).
Branch On Not Equal To Zero	BNEZ *rx*, *offset*	If *rx* ≠ 0, the program branches to the target address specified as a 8-bit *offset* relative to PC + 2 (or PC + 4 when EXTENDed).
Branch On T8 Equal To Zero	BTEQZ *offset*	If t8 = 0, the program branches to the target address specified as a 16-bit *offset* relative to PC + 2 (or PC + 4 when EXTENDed).
Branch On T8 Not Equal To Zero	BTNEZ *offset*	If t8 ≠ 0, the program branches to the target address specified as a 16-bit *offset* relative to PC + 2 (or PC + 4 when EXTENDed).
Unconditional Branch	B *offset*	The program unconditionally branches to the target address specified as a 16-bit *offset* relative to PC + 2 (or PC + 4 when EXTENDed).
* Branch And Link	BAL *offset*	The program unconditionally branches to the target address specified as a 16-bit *offset* relative to PC + 2 (or PC + 4 when EXTENDed). The address of the instruction following the delay slot is saved in r31.

* Enhancements from the TX19 to the TX19A

Table 4-27 Special Instructions (16-Bit ISA)

Instruction	Format	Operation
Breakpoint	BREAK *code*	A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.
Software Debug Breakpoint Exception	SDBBP *code*	A debug breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.
* Disable Interrupt	DI	The IE bit in the Status register is cleared.
* Enable Interrupt	EI	The IE bit in the Status register is set.
* System Call	SYSCALL *code*	A System Call exception occurs, immediately and unconditionally transferring control to the exception handler.
* Synchronize	SYNC	The instruction pipeline is interlocked until any load or store fetched before the current instruction is completed.
* Exception Return	ERET	If the ERL bit in the Status register is set, the PC is restored from the Error PC register. Otherwise, the PC is restored from the EPC register
* Debug Exception Return	DERET	Program control is transferred back to a User program from a debug exception handler. The return address in the DEPC register is restored into the PC.
* Enter Standby Mode	WAIT	If the RP bit in the Status register is set, the processor enters DOZE mode. If the RP bit is cleared, the processor enters HALT mode.

* Enhancements from the TX19 to the TX19A

Chapter 5 CPU Pipeline

5.1 Architecture Overview

As described in Section 2.5, *Pipeline Architecture*, the processing of an instruction is broken down into a sequence of simpler suboperations. Because tasks required to process an instruction are fragmented, an instruction does not need the entire hardware resources of the execution unit. Each suboperation is performed by a separate hardware section called a *stage*, and each stage passes its result to a succeeding stage. The TX19A pipeline has five stages, Fetch (F), Decode (D), Execute (E), Memory Access (M) and Register Write-back (W). For example, after an instruction completes the D stage, it can proceed to the E stage while the subsequent instruction can advance into the D stage. Each of the five pipe stages requires approximately one clock cycle. Therefore, once the pipeline has been filled, the execution of five instructions is overlapped at a time, as shown in Figure 5-1.

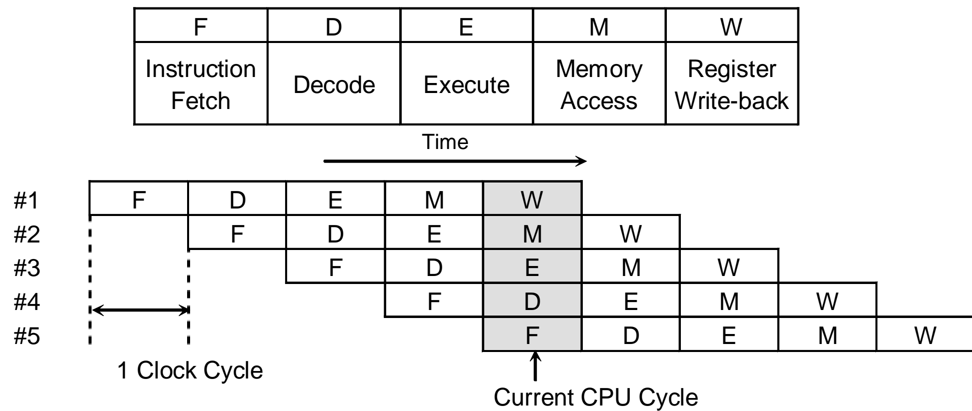


Figure 5-1 Five CPU Pipeline Stages

The following paragraphs describe the operations in each stage that occur for the most-commonly used instructions.

Instruction Fetch (F): In this stage, the instruction is fetched from the instruction memory subsystem (i.e., instruction ROM or instruction RAM). Instructions are fetched in one-word units, whether in 16-bit or 32-bit ISA mode.
Decode (D): During this stage, the instruction is decoded and required operands are read from the on-chip register file.

Execute (E): In this stage, one of the following occurs:

- The arithmetic logic unit (ALU) starts the integer arithmetic, logical or shift operation.
- For load and store instructions, the ALU initiates the bus cycle and calculates the effective address by adding the offset value to the contents of the base register at the same time.
- For jump instructions, the ALU calculates the jump target address.

- For branch and branch-likely instructions, the ALU determines whether the branch condition is true and calculates the branch target address.

Memory Access (M): For loads and stores, data memory is accessed.

Register Write-back (W): In this stage, one of the following occurs:

- The results of the ALU operation during the E stage is written back to the on-chip register file.
- If the instruction is a jump-and-link, branch-and-link or branch-likely-and-link, the return address is written to register r31 (ra).

In a pipelined machine like the TX19A, there are certain instructions that can potentially disrupt the smooth advance through the pipeline. This problem is referred to as *pipeline hazards*. The sections that follow describe when pipeline hazards occur and how they are handled by hardware and software.

5.2 Load, Store and SYNC Instructions

The performance of software systems is drastically affected by how well software designers, especially assembly-language programmers, understand the basic hardware technologies at work in the processor. This section describes load delays, non-blocking loads, shared memory synchronization and so on from the view point of the CPU pipeline.

5.2.1 Load Delays

Figure 5-2 illustrates how the load instruction advances through the CPU pipeline.

F	D	E	M	W
Instruction Fetch	Decode	Effective Address Calculation & Bus Cycle Initiation	Memory Access	Resister Write-back

Figure 5-2 Load Instruction

Load instructions read an operand from memory into a CPU register for a subsequent operation by other instructions. In the case of loads from the on-chip fast memory, an operand becomes available after completion of the Memory Access (M) stage of the load instruction because it is internally forwarded at the M stage before the Register Write-back (W) stage. Still, the operand is not immediately usable for the Execute (E) cycle of the subsequent instruction, as shown in Figure 5-3.

This is called *data dependency*. In Figure 5-3, the TX19A handles data dependency by inserting a wait (or "*stall*") cycle into the E stage of the next instruction. Figure 5-3 depicts a delay (or latency) of one cycle. The instruction that immediately follows the load instruction is said to be in the *load delay slot*. Loads from external memory incur additional stall cycles.

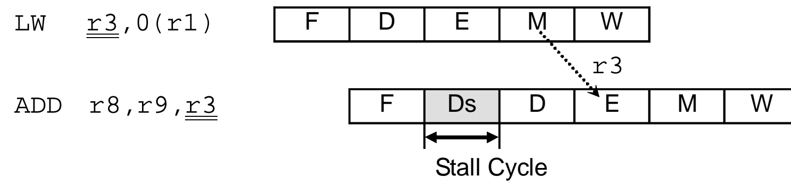
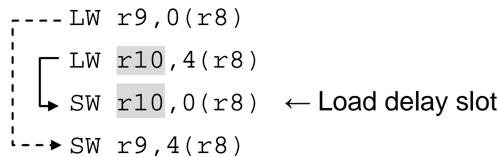


Figure 5-3 Data Dependency Resulting from a Load Instruction

However, this is not a very efficient use of the pipeline. The optimizer, which is executed as a part of the compiler or assembler, can rearrange the code to ensure that the instruction in the load delay slot does not require the operand loaded by the previous load instruction. Figure 5-4 gives an example of re-ordering instructions to remove data dependency. This is a part of the code to swap the contents of two memory locations.

- With data dependency



- Without data dependency

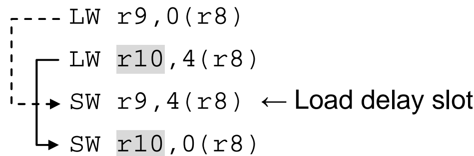


Figure 5-4 Re-ordering Instructions to Remove Data Dependency

In the rearranged code, the SW instruction does not depend on the availability of data from the immediately preceding LW instruction. Therefore, the load delay slot for "LW r10, 4 (r8)" can be filled with a useful instruction, "SW r9, 0(r8)," so that the pipeline is fully utilized.

5.2.2 Non-blocking Loads

If the instruction that immediately follows a load instruction does not access the target register (*rt*) of the load instruction, data dependency does not occur. The TX19A recognizes the presence of data dependency, and if there is no data dependency, it continues to execute subsequent instructions. This is called *non-blocking loads*. By virtue of non-blocking loads, external memory accesses do not stall the CPU pipeline. All the other parts of the pipeline can continue to work on non-dependent instructions while external memory is being accessed.

In Figure 5-5 below, the TX19A continues to execute independent instructions (ADD, r6, r4, r2 and ADD r7, r5, r2) without stalling on the external memory access resulting from the LW Instruction. It defers execution of a dependent instruction (ADD, r8, r9, r3) until the data has been returned.

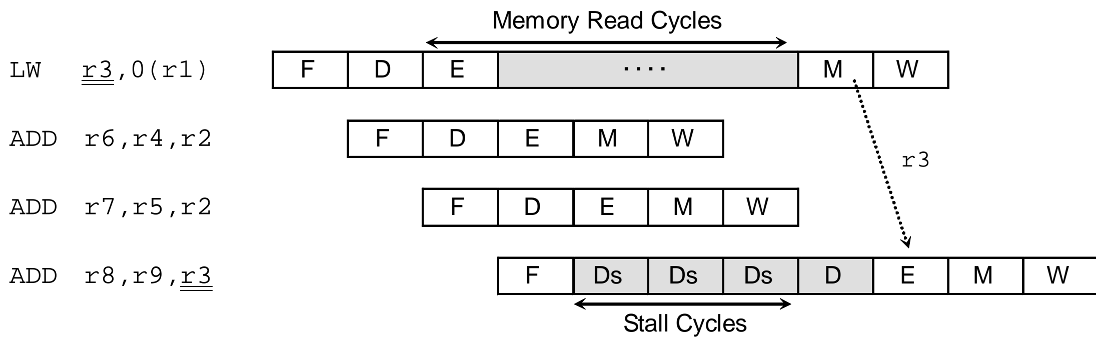


Figure 5-5 Non-blocking Loads

The non-blocking load capability of the TX19A allows the optimizing compiler to rearrange the code to "prefetch" data from memory before a need actually arises to reference it. Selective use of prefetches based on the compiler's optimization can yield significant performance improvement.

5.2.3 Store Instructions (32 Bit ISA/ 16 Bit ISA)

Figure 5-6 illustrates how the store instruction advances through the CPU pipeline.

F	D	E	M	W
Instruction Fetch	Decode	Effective Address Calculation & Bus Cycle Initiation	Mempory Access in WAIT	—

Figure 5-6 Store Instruction

Stores to the on-chip fast memory occur during the Memory Access (M) stage; no operation occurs in the Register Write-back (W) stage. Stores to external memory take more than one cycle.

The store instruction is to store the data in CPU register to the memory. Figure 5-7 shows how to store to the on-chip fast memory. Stores to the on-chip fast memory occur during the Effective Address Calculation (E) stage and they take 1 clock as a write bus cycle. No operations occur in the Memory Access stage and the Register Write-back stage.
Figure 5－8 shows the procedure to execute the instruction on the external memory access. Stores to the external memory take more than 2 clocks as a write bus cycle. With the TX19A, the pipelines never stall by the continuous memory access instructions because its on-chip write buffer can store 4 write-data at the maximum. The instructions using subsequent write buffer stall when the write buffer space is full.

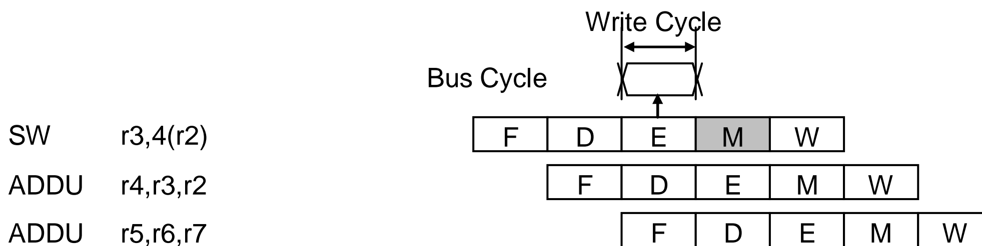


Figure 5-7 Access to the On-chip Fast Memory

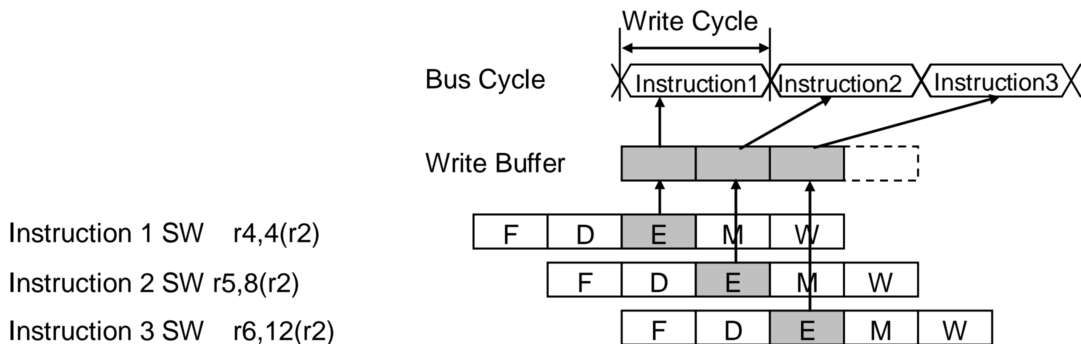


Figure 5-8 Continuous Access to the External Memory

5.2.4 SYNC Instruction (32 Bit ISA/ 16 Bit ISA)

Load and store instructions execute memory accesses during the M stage. In the meantime, the TX19A continues to execute other instructions in parallel.

Figure 5-9 illustrate the SYNC instruction procedure. The SYNC instruction provides an ordering function for the effects of load/store and subsequent instructions. The SYNC instruction ensures that all loads and stores initiated prior to this instruction are completed before any instruction after this instruction is allowed to start. To enforce in-order execution, stall cycles are inserted into the M stage until the previously initiated loads and stores are completed.

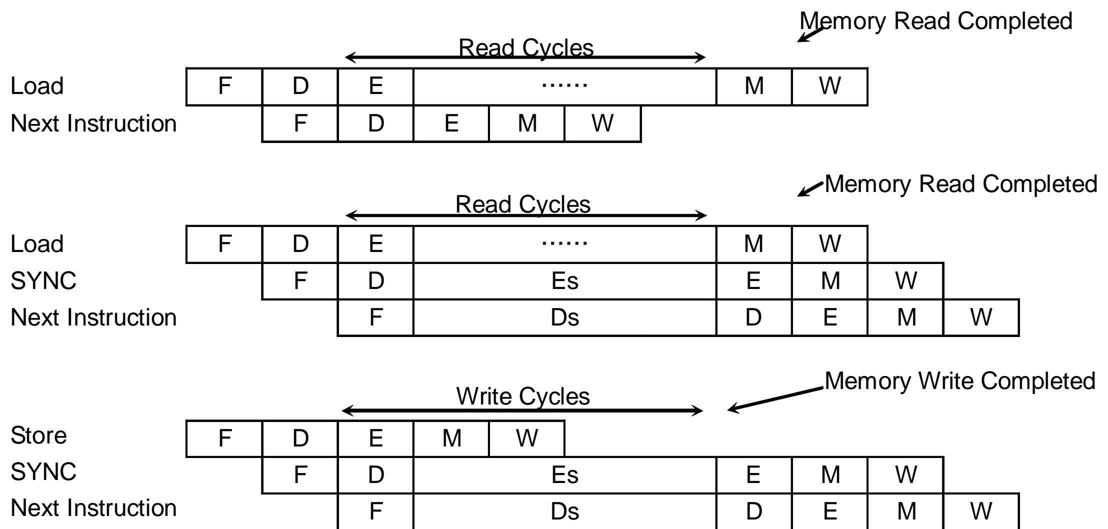


Figure 5-9 SYNC Instruction

5.2.5 Bit Manipulation Instruction (16 Bit ISA)

There are two kinds of the Bit Manipulation Instructions. One is to place the result back into the memory. Another is to place the result back into the CPU register. Figures 5-10 and 5-11 show each procedure.

F	D	E	M	W
Instruction Fetch	Decode	Effective Address Calculation & Bus Cycle Initiation	Memory Access	–

Figure 5-10 Placing the result back into the memory

F	D	E	M	Md	W
Instruction Fetch	Decode	Effective Address Calculation & Bus Cycle Initiation	Memory Access	Computation	Register Write-back

Figure 5-11 Placing the result back into the CPU register

As for the instruction illustrated in Figure 5-10, the write buffer enters the bus operation without any pipeline stall same as the store instruction. See Figure 5-12 for the detailed procedure.

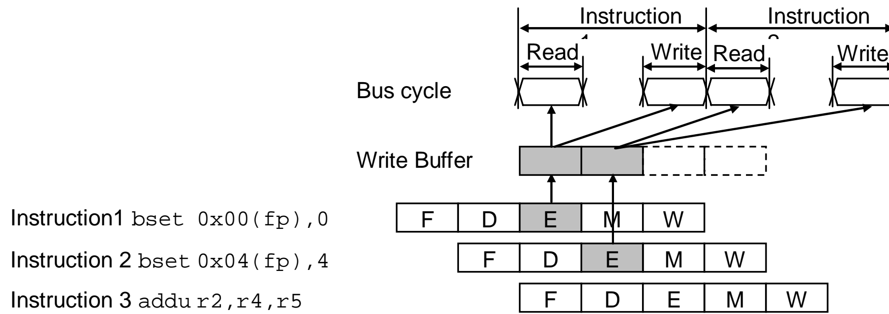


Figure 5-12 Detailed procedure of placing the result back into the memory

The instruction shown in Figure 5-11 has the different operations depend on the contents.
The pipelines do not stall during the subsequent instructions if the instruction immediately after the bit manipulation instruction does not access the target register (t8) of the bit manipulation instruction. At that time, the bit manipulation instructions, BTST and BEXT, are operating in non-blocking load. On the other hand, the pipelines stall during the subsequent instructions if the instruction immediately after the bit manipulation instruction accesses the target register (t8).
Figure 5-13 shows the detailed procedure of the case with CPU. The two ADDU instructions do not access the target register. Therefore, these two immediately after the BTST can be executed without stall. The fourth BTEGZ, however, stalls since it needs to refer to the target register.

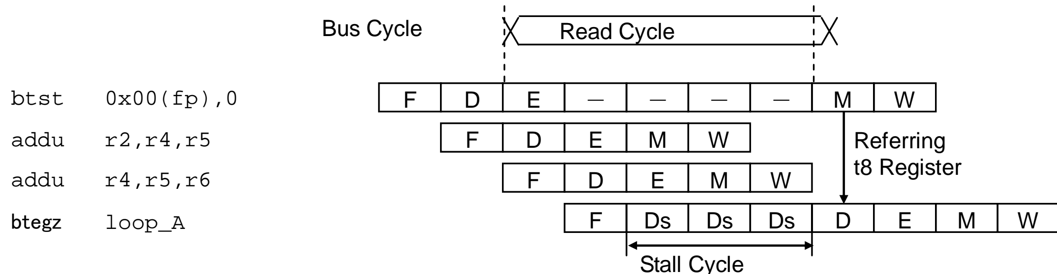


Figure 5-13 Detailed procedure of placing the result back into the CPU register

In non-blocking load, the W stage of the Bit Manipulation Instruction may conflict with the W stage of the subsequent instruction. Then the subsequent instructions stall.

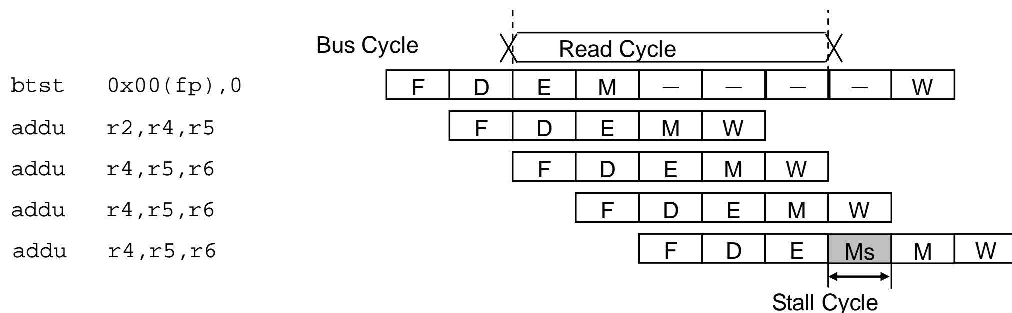


Figure 5-14 W Stages Conflict

5.3 Jump, Branch and Branch-Likely Instructions

Jump and branch instructions involve a delay or latency of two instruction cycles. This section explains how this latency is reduced to one cycle by software intervention. This section also describes how branch-likely instructions are processed through the pipeline.

5.3.1 Jump and Regular Branch Instructions (32-Bit ISA)

Figure 5-15 shows how jump and regular branch instructions advance through the CPU pipeline.

F	D	E	M	W
Instruction Fetch	Decode	Target Address Calculation & Branch Condition Test PC Update	No Operation	Register Write-back

Figure 5-15 Jump and Branch Instructions

For jump and branch instructions, one of the following occurs in the Execute (E) stage:

- For jump instructions, the ALU calculates the jump target address.
- For branch and branch-likely instructions, the ALU determines whether the branch condition is true and calculates the branch target address.

No operation is performed in the M stage. If the instruction is a jump-and-link or a branch-and-link, a return address is written to register r31 (ra) in the Register Write-back (W) stage.

See Figure 5-16 for the illustrated regular branch instruction. The jump or branch target address becomes available during the E stage. A jump or branch occurs with a delay of two instructions cycles since the fetch of the target instruction occurs after the target address calculation. the instruction in the delay slot occurs immediately after the jump or regular branch instruction is always executed prior to the jump/branch taking effect. Therefore, the delay cycle caused by the jump or branch instruction looks as if only 1 cycle. It is the responsibility of the compiler to rearrange the code to fill a jump or branch delay slot with a useful instruction. If there is no useful instruction, the compiler must fill the delay slot with a NOP.

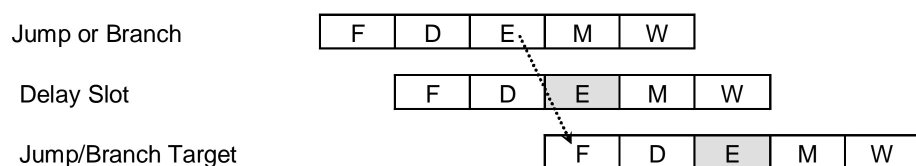


Figure 5-16 Jump and Branch Delay Slots

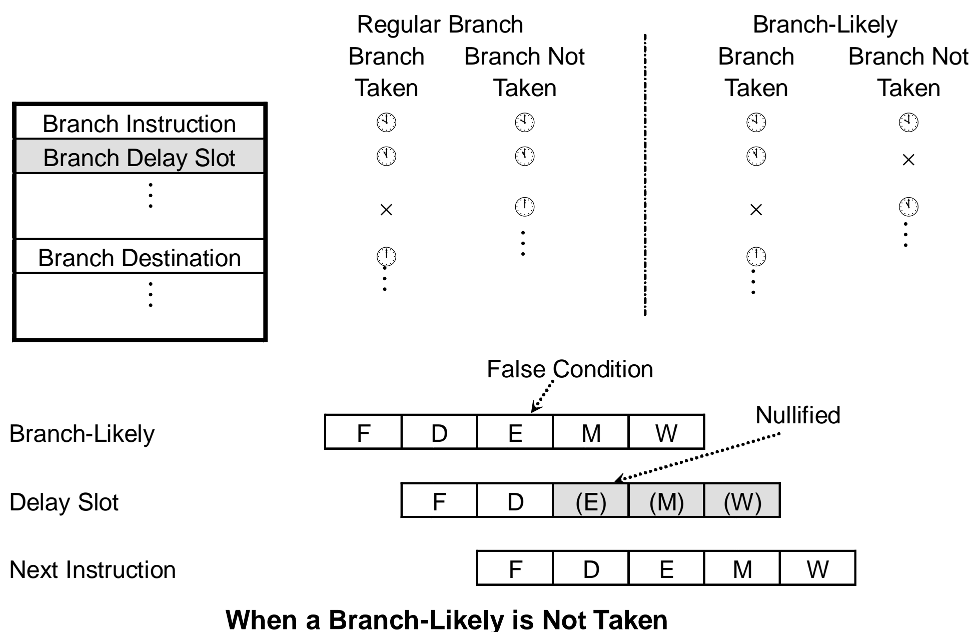
Note 1: Please do not fill a jump or branch delay slot with a jump or branch instruction to avoid instable hardware operation.

Note 2: Please do not fill a branch delay slot with any instruction may cause an effect to program logic since the regular branch instruction always executes the one in the delay slot regardless of whether the branch is taken or not.

5.3.2 Branch-Likely Instructions (32-Bit ISA)

A regular branch instruction causes the TX19A to always execute the instruction in a branch delay slot, regardless of whether the branch is to be taken or not. Therefore, the instruction in the branch delay slot must logically precede the branch instruction. for the difference.

On the other hand, a branch-likely instruction causes the TX19A to nullify the instruction in the delay slot at the Execute (E) stage if the branch is not taken. If a branch is taken, the instruction in the delay slot is executed. This approach allows the compiler to fill a branch delay slot with the branch target instruction (see Figure 5-17).



When a Branch-Likely is Not Taken

Figure 5-17 Branch-Likely Instruction

5.3.3 Jump Instructions (16-Bit ISA)

The JAL and JALX instructions in the 16-bit ISA are still 32-bits wide; so in 16-bit ISA mode, the previous TX19 needs to execute a jump instruction in two steps as shown in Figure 5-18. The TX19 performs no operation during the first D and E stages. Instead it waits for the second half of the instruction code to come in order to calculate the effective address of the jump destination. This address calculation occurs in the E stage of the second half of the jump instruction. As a consequence, jump instructions in the 16-bit ISA occur with a two-instruction delay.

In contrast, the TX19A fetches and decodes a jump instruction in one go, thereby reducing a jump delay from two instruction cycles to one (see Figure 5-19).

Note: Please do not fill a jump delay slot with a jump or branch instruction to avoid instable hardware operation.

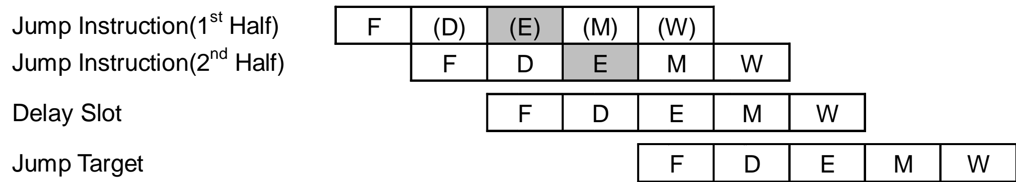


Figure 5-18 Jump Instruction (TX19 16-Bit ISA)

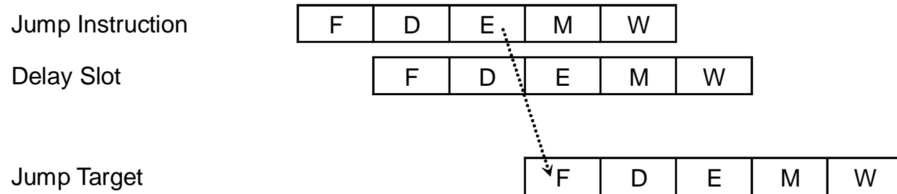


Figure 5-19 Jump Instruction (TX19A 16-Bit ISA)

5.3.4 Branch Instructions (16-Bit ISA)

Unlike the 32-bit ISA, the 16-bit ISA has no delayed branches (see Figure 5-20). The branches take effect before the next instruction. Thus if the branch is taken, the following instructions are not executed. For this reason, any instruction can be placed immediately after a branch instruction.

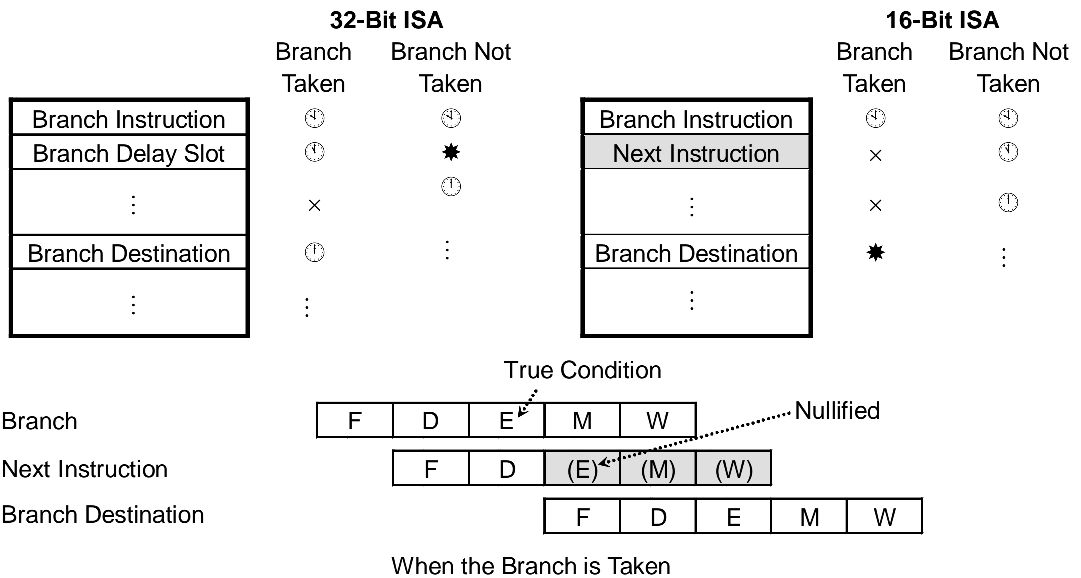


Figure 5-20 Branch Instruction (16-Bit ISA)

5.3.5 SAVE ・ RESTORE Instructions (16-Bit ISA)

One SAVE/RESTORE instruction can save or restore the data in multiple registers.
See figure 5-21 and 5-22 for the details.
The next instructions stall until the contents of the stack pointer register (r29) are rewritten with the final data restore or save.

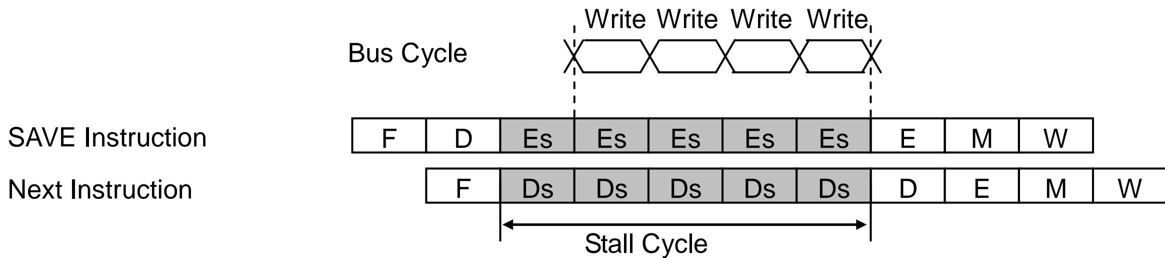


Figure5-21 SAVE Instructions

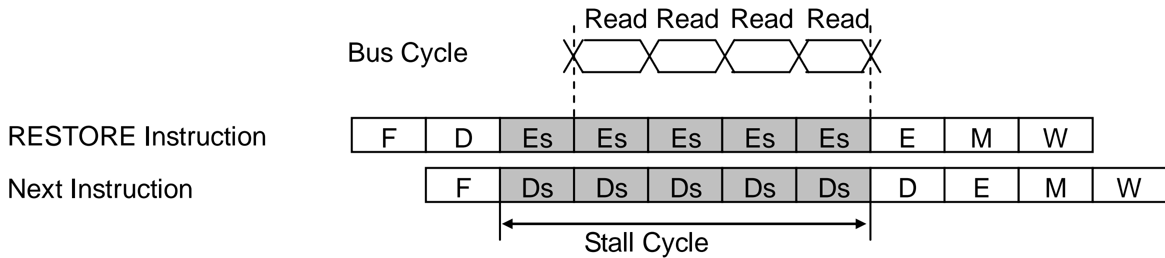


Figure 5-22 RESTORE Instructions

5.4 Divide Instructions

Any integer divide instruction is transferred to the dedicated divide unit as remaining instructions continue through the pipeline. The divide unit keeps running even when delay cycles and exceptions occur. The quotient and the remainder of the divide instruction are saved in the LO and HI registers.

The TX19A starts a divide operation in the E stage; it takes 35 cycles for the divide operation to complete, independent of the magnitude and sign of the operands. If the divide instruction is followed by an MFHI, MFLO, MADD, MADDU, MSUB or MSUBU instruction before the quotient and the remainder are available, the pipeline stalls until they do become available.

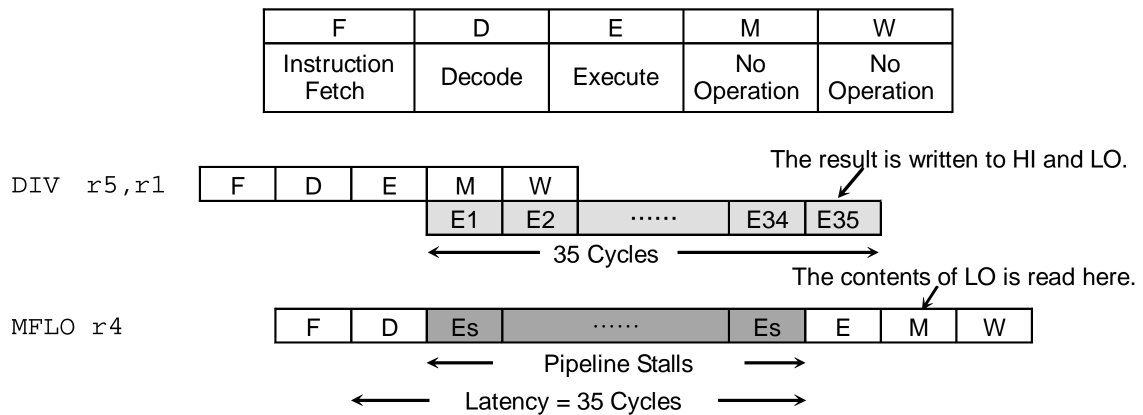


Figure 5-23 Divide Instructions

5.5 Multiply, Multiply-and-Add and Multiply-and-Subtract Instructions

Any integer multiply, multiply-and-add and multiply-and-subtract instructions are transferred to the dedicated MAC unit as remaining instructions continue through the pipeline. It takes a single cycle for a multiply, multiply-and-add or multiply-and-subtract instruction to complete.

Because it takes only one cycle for a multiply, multiply-and-add or multiply-and-subtract instruction to complete the E stage, multiple multiply, multiply-and-add and multiply-and-subtract instructions can be executed back-to-back without causing pipeline stalls (see Figure 5-24).

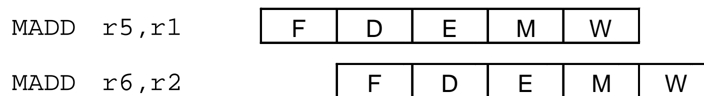


Figure 5-24 Back-to-Back Multiply-and-Add Instructions

The MFHI and MFLO instructions read the contents of the HI and LO registers. Multiply, multiply-and-add and multiply-and-subtract instructions can be followed by an MFHI or MFLO instruction without causing pipeline stalls (see Figure 5-25).

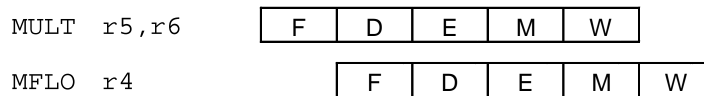


Figure 5-25 Multiply Instruction Followed by an MFLO Instruction

Remember that the result of the multiply, multiply-and-add and multiply-and-subtract instructions becomes available after completion of the M stage instead of the E stage. If the multiply, multiply-and-add or multiply-and-subtract instruction specifies a general-purpose register as a destination register (*rd*), subsequent instructions should not access that register until the result is saved in *rd*. Otherwise, the pipeline stalls at the D stage until it does become available.

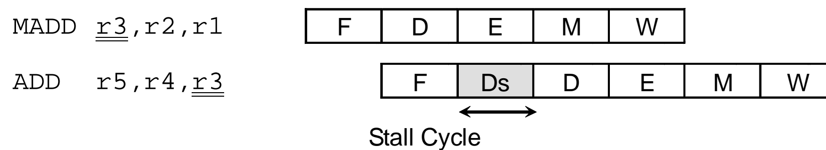


Figure 5-26 Structural Hazard Involving a Multiply Instruction

5.6 EXTENDed Instructions (16-Bit ISA)

The EXTEND prefix turns 16-bit instructions in the 16-bit ISA into 32 bits. The machine code of an EXTENDed instruction consists of an 16-bit EXTEND code and the 16-bit instruction code that is to be EXTENDed. While the TX19 executes any EXTENDed instruction in two steps (Figure 5-28), the TX19A improves instruction throughput by executing each EXTENDed instruction in one go (Figure 5-27).

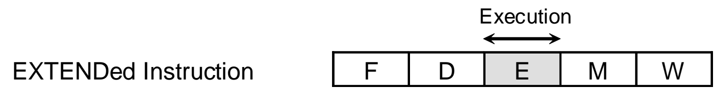
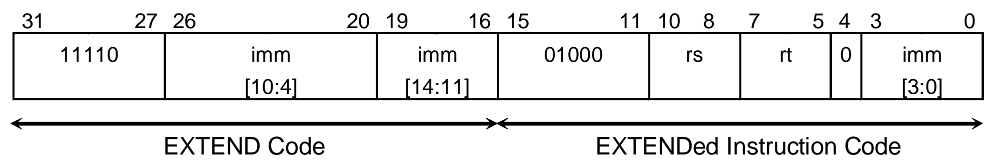


Figure 5-27 EXTENDed Instruction (TX19A 16-Bit ISA)



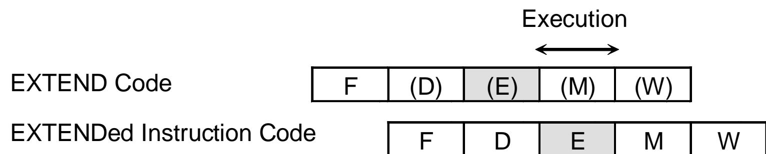


Figure 5-28 EXTENDed Instruction (TX19 16-Bit ISA)

Chapter 6 Memory Management

This chapter describes the operating modes of the TX19A processor, the virtual and physical address spaces and how they are mapped.

6.1 Operating Modes

The TX19A has two modes of operation, User mode and Kernel mode. The TX19A enters Kernel mode whenever an exception is taken. Since a Reset exception occurs when a system is reset, the TX19A wakes up in Kernel mode. The processor switches to User mode when the ERET (Exception Return) or DERET (Debug Exception Return) instruction is executed.

■ User Mode

The operating mode determines the addresses, registers and instructions that are available to a program. The use of them is restricted under User mode. While the processor is operating in User mode, it is permitted to access a linear address space of 2 GB (kuseg) starting at virtual address 0x0000_0000. The CP0 registers are accessible only when the CU0 bit in the Status register is 1.

When the processor is operating in User mode, both of the following conditions are true: 1) the UM bit in the Status register is set; and 2) the ERL and EXL bits in this register are cleared.

■ Kernel Mode

Kernel mode has higher privileges than User mode. Kernel-mode programs are permitted to use all addresses, registers and instructions. Operating system routines, general exception handlers and debug exception handlers are executed in Kernel mode.

When the processor is operating in Kernel mode, any of the following conditions is true: 1) the DM bit in the Debug register is set; 2) the UM bit in the Status register is cleared; 3) the ERL bit in the Status register is set; or 4) the EXL bit in the Status register is set.

Note: TX19A only allows using Kernel mode.

6.2 Virtual Address Segments

Figure 6-1 shows the virtual address segments available in User and Kernel modes. While the processor is operating in User mode, a single, uniform virtual address space (kuseg) of 2 GB is available. While the processor is operating Kernel mode, four distinct virtual address segments, kuseg, kseg0, kseg1 and kseg2, are simultaneously available.
Each segment is architecturally predefined as cached or uncached; however, because the TX19A

does not have a cache on-chip, cacheability has no meaning.

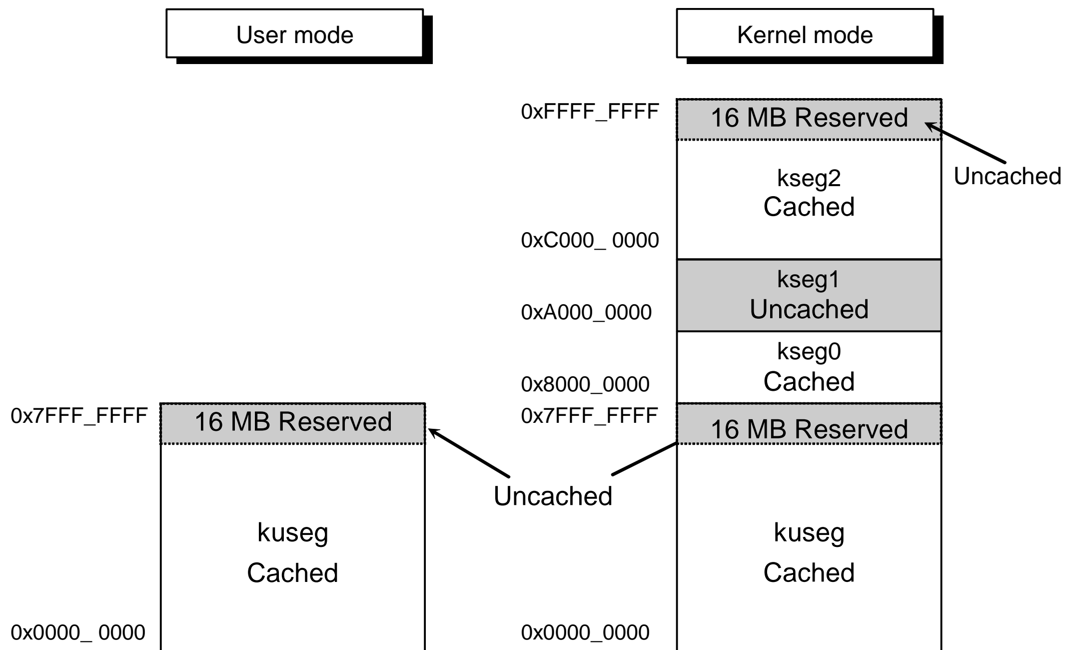


Figure 6-1 Virtual Address Segments

■ Kuseg (Kernel/User Segment)

Kuseg is a 2-GB segment designed to be used by User-mode programs while providing accessibility in Kernel mode. This virtual address space begins at address 0x0000_0000 and runs up to 0x7FFF_FFFF; so all valid User-mode virtual addresses have the most-significant bit cleared to 0. A User program attempt to reference a Kernel address with the most-significant bit set to 1 causes an Address Error exception. The upper 16 MB of kuseg should not be used. This region is reserved for on-chip resources which map to these virtual addresses.

■ Kseg0, kseg1 and kseg2 (Kernel Segments)

The virtual address space accessible only in Kernel mode consists of three distinct segments called kseg0, kseg1 and kseg2, which total 2 GB in size. The Kernel segments start at virtual address 0x8000_0000 and run up to
0xFFFF_FFFF.

- Kseg0 is a 512-MB segment, beginning at virtual address 0x8000_0000; all references through this segment are cacheable.
- Kseg1 is also a 512-MB segment, beginning at virtual address 0xA000_0000, but unlike kseg0, all references through this segment are uncacheable.
- Kseg2 is a 1-GB linear address space, beginning at virtual address 0xC000_0000. The upper 16 MB of kseg2 should not be used. This region is reserved for on-chip resources which map to these virtual addresses; 2-MB addresses from 0xFF20_0000 to 0xFF3F_FFFF are reserved for

debugging. While the upper 16 MB is uncacheable, the remaining region of kseg2 is cacheable.

6.3 Address Translation

The virtual-to-physical address translation is done through a direct segment mapping, which allows Kernel-mode software to be protected from User-mode accesses without requiring virtual page management software. Direct segment mapping of virtual-to-physical addresses is illustrated in Figure 6-2.

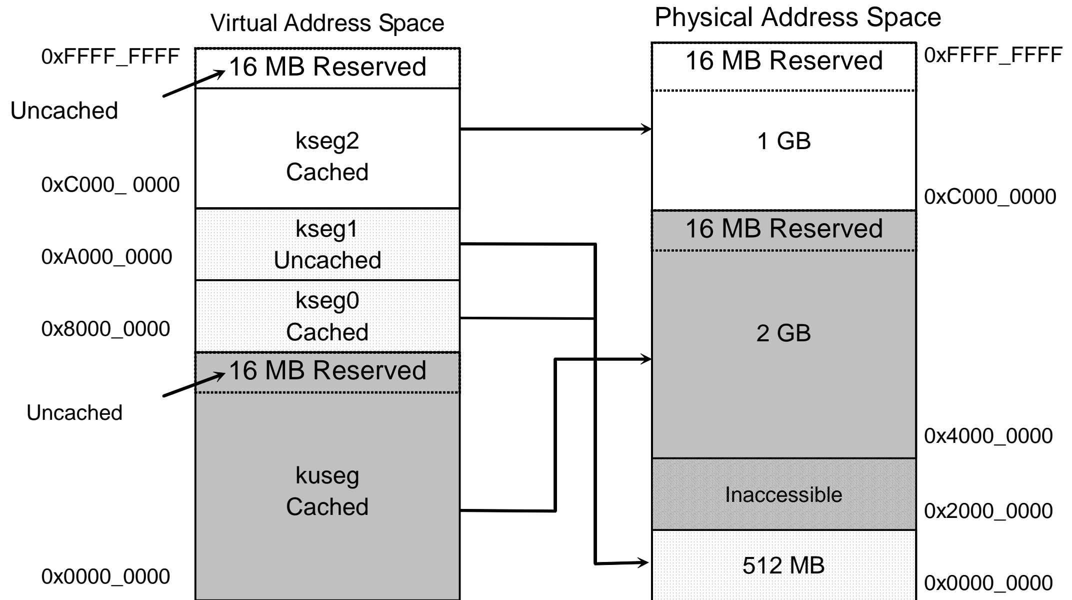


Figure 6-2 Virtual to Physical Address Translation

Figure 6-3 shows the virtual address format used by the TX19A. The three highest bits represent segment numbers; only these three bits are involved in virtual-to-physical address translation.

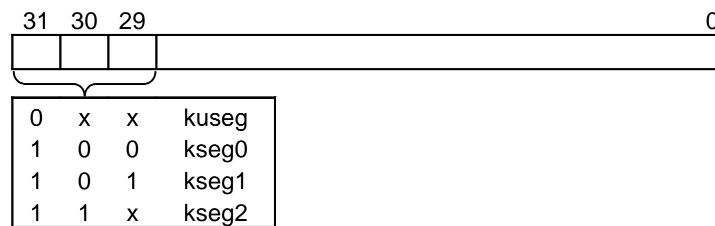


Figure 6-3 Virtual Address Format

- Kuseg is mapped to a contiguous 2-GB region of the physical address space starting at 0x4000_0000. The physical address is constructed by replacing "0x" in the two highest-order bits with "01."
- Virtual addresses in both kseg0 and kseg1 are mapped to the 512-MB physical address space starting at address 0x0000_0000. When the three highest-order bits of the virtual address are "100," that virtual address resides in kseg0. When the three highest-order bits of the virtual

address are "101," that virtual address resides in kseg1. The physical address is constructed by replacing these three bits with "000."

- Virtual addresses in kseg2 are directly output as physical addresses.

Table 6-4 Segment Mapping from Virtual to Physical Addresses

Segment		Virtual Addresses	Physical Addresses	Cacheability	Operating Mode
kseg2	Reserved	0xFF20_0000~0xFFFF_FFFF	0xFF00_0000~0xFFFF_FFFF	Uncacheable	Kernel
	Free	0xC000_0000~0xFEFF_FFFF	0xC000_0000~0xFEFF_FFFF	Cacheable	Kernel
kseg1		0xA000_0000~0xBFFF_FFFF	0x0000_0000~0x1FFF_FFFF	Uncacheable	Kernel
kseg0		0x8000_0000~0x9FFF_FFFF	0x0000_0000~0x1FFF_FFFF	Cacheable	Kernel
kuseg	Reserved	0x7F00_0000~0x7FFF_FFFF	0xBF00_0000~0xBFFF_FFFF	Uncacheable	Kernel/ User
	Free	0x0000_0000~0x7EFF_FFFF	0x4000_0000~0xBEFF_FFFF	Cacheable	Kernel/ User

It is prohibited to place programs across two segments. Jumps and branches must not transfer program control outside the current segment.

Chapter 7 Internal I/O Bus Operation

7.1 Internal Memory Interface

Figure 7-1 shows an example of the bus interface inside the TX19A core. To maximize performance, the TX19A implements a Harvard architecture, wherein there are two separate sets of address and data buses for code (instructions) and data (operands). Additionally, the TX19A allows very fast access to the on-chip memory – one word of data per clock cycle. Consequently, an execution rate of one instruction for each clock cycle is achieved.

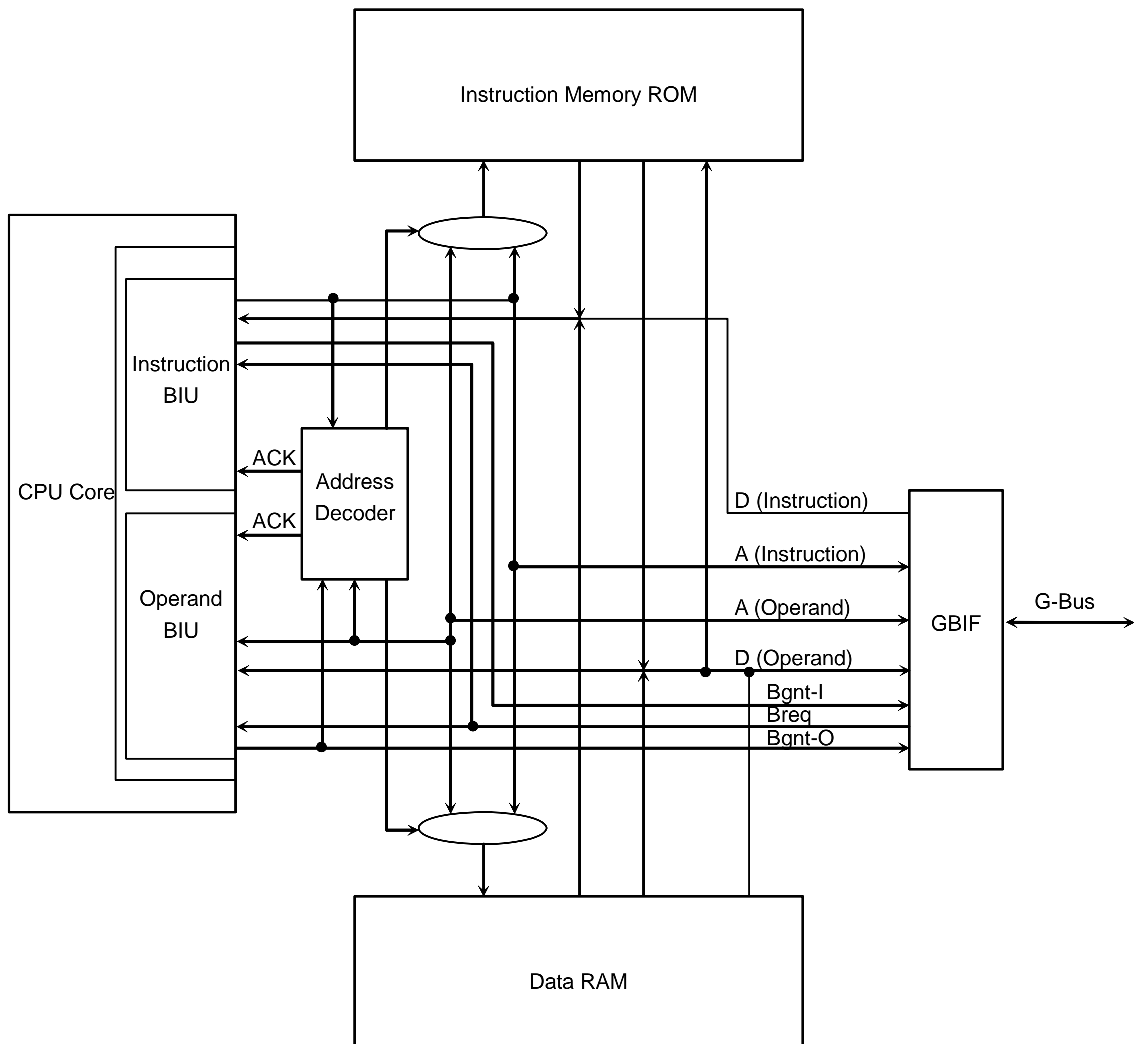


Figure 7-1 General Internal Memory Interface

7.2 Operand Read and Instruction Fetch Operations

Figure 7-2 and Figure 7-3 show the bus cycle timing for operand reads and instruction fetches. The TX19A core features pipelined addressing where it allows up to two outstanding bus cycles at any given time. While the TX19A core waits for the data for the first bus cycle, the address for a second bus cycle is issued. Using pipelined addressing, the TX19A provides support for zero-wait-state reads even for relatively slow memories like flash.

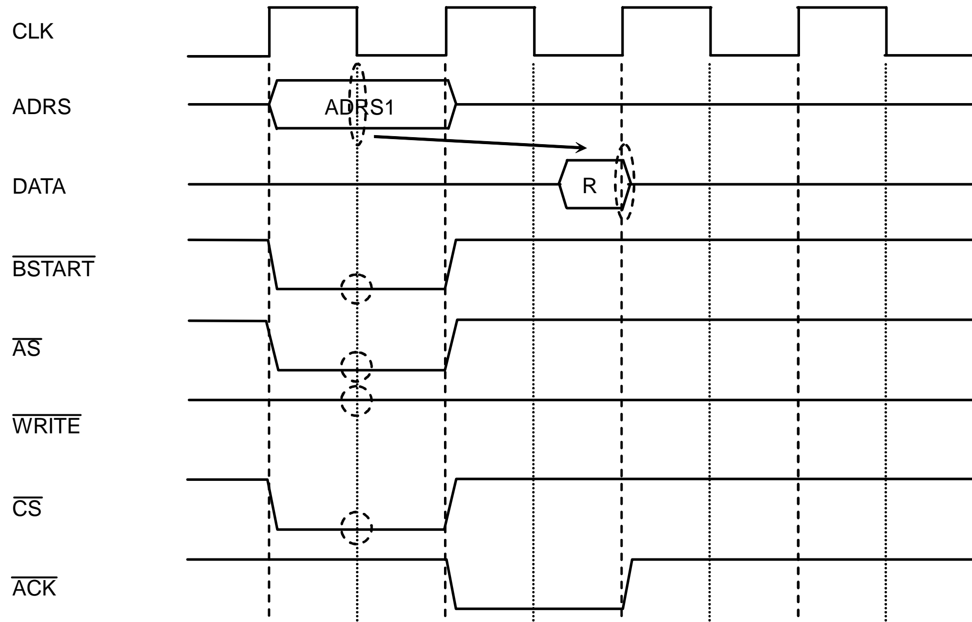


Figure 7-2 Memory Read Timing (Zero-Wait State)

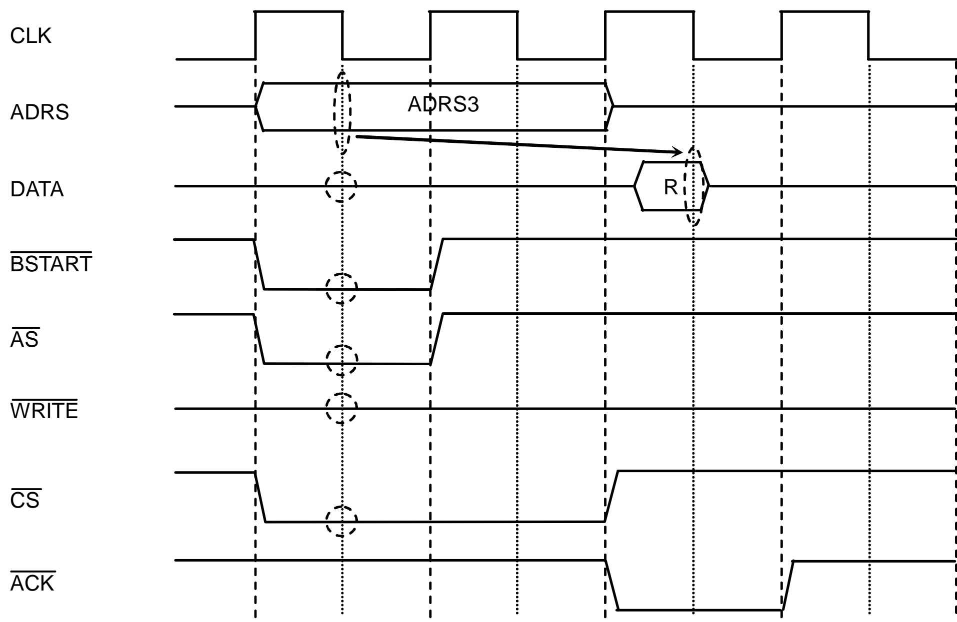


Figure 7-3 Memory Read Timing (1 Wait State for ADRS3)

7.3 Write Operation

Basically, memory write cycles use much the same protocol as memory read cycles. The TX19A core drives out a memory address on the falling edge of the system clock. At the same time, Byte Enable, Bus Start (BSTART*), Address Strobe (AS*), Write (WRITE*) and Chip Select (CS) etc. are also asserted.

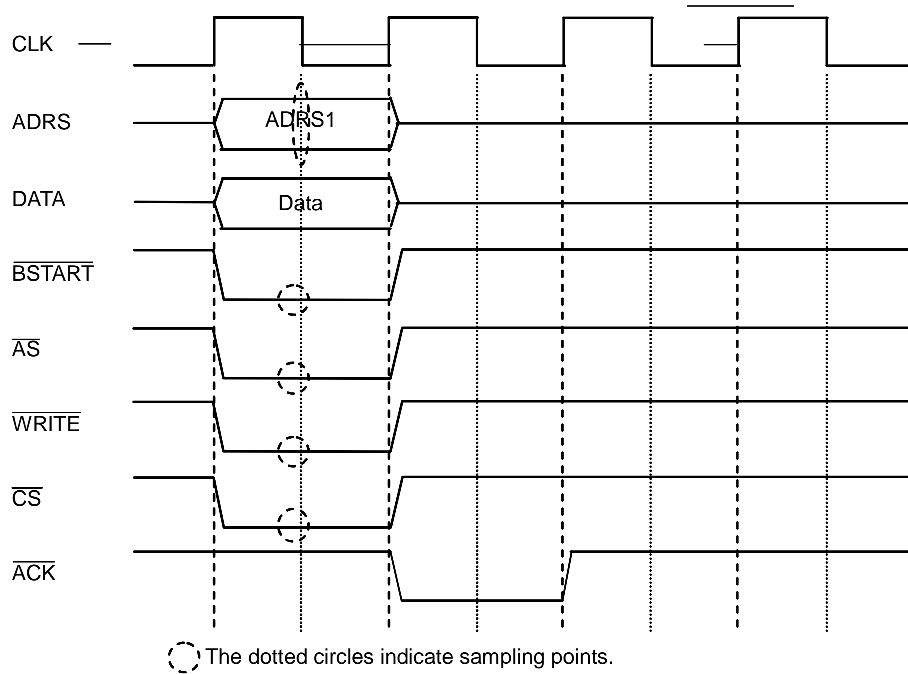


Figure 7-4 Write Timing (Zero-Wait State)

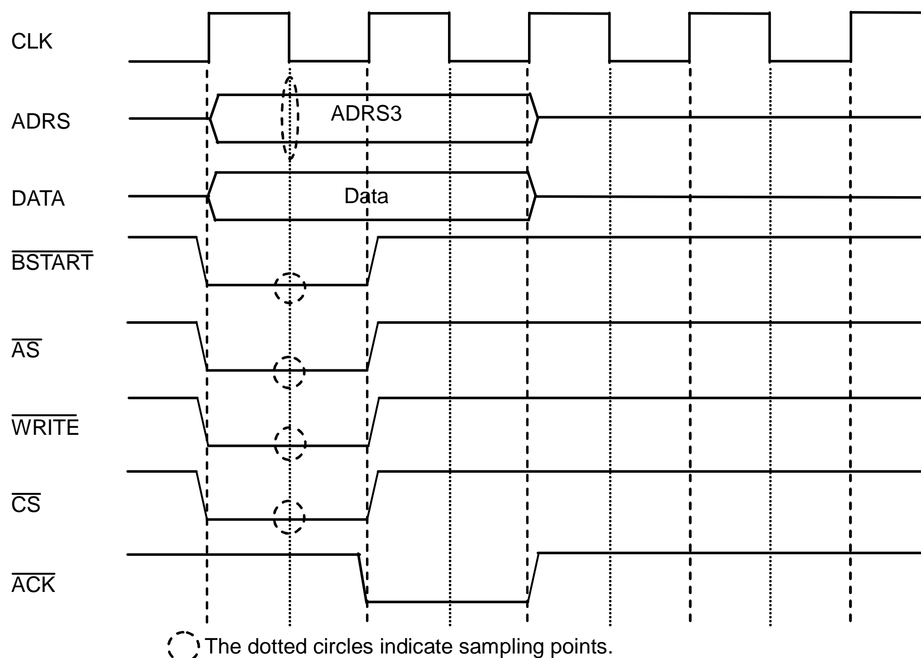


Figure 7-5 Write Timing (1 Wait State for ADR3)

Chapter 8 System Control Coprocessor (CP0)Registers

This chapter describes the system control coprocessor (CP0) registers used for system configuration, memory management and exception processing.

When the processor is in Kernel mode, the system control coprocessor instructions can always use the CP0 registers. When the processor is in User mode, the CP0 registers are accessible only when the CU0 bit in the Status register is set.

8.1 Overview

Table 8-1 provides a brief description of each of the CP0 registers. Register numbers are used by software when issuing the Move From CP0 (MFC0) and Move To CP0 (MTC0) instructions.

Table 8-1 CP0 Registers

Category	Register Name	Register Number	Description
System Configuration	Config Config1 Config2 Config3	16 (SEL0) 16 (SEL1) 16 (SEL2) 16 (SEL3)	Specifies various configuration options for the TX19A processor.
General Exception Handling	BadVAddr	8 (SEL0)	Displays the most recent virtual address that caused a virtual-to-physical address translation error. Read-only.
	Count	9 (SEL0)	Acts as a timer, incrementing at a constant rate.
	Compare	11 (SEL0)	Maintains a constant value compared against the Count register value.
	Status	12 (SEL0)	Contains operating mode (User/Kernel), interrupt enable and other states of the processor.
	Cause	13 (SEL0)	Displays the cause of the last exception.
	EPC	14 (SEL0)	Contains the upper 31 bits of the address of the exception-causing instruction, from which point processing resumes after the exception has been serviced, combined with the ISA mode bit that was in effect before the exception occurred.
	ErrorEPC	30 (SEL0)	Similar to the EPC register except that ErrorEPC is used on Reset and NMI exceptions.
	PRId	15 (SEL0)	Contains the revision identifier of the TX19A processor. Read-only.
	IER	9 (SEL7)	Manipulates the interrupt enable/disable bit in the Status register.
	SSCR	22 (SEL0) ／9 (SEL6)	Contains a two-level stack (current and previous) for the shadow register set used.
Debug Exception Handling	Debug	23 (SEL0)	Displays the cause and the current status of a debug exception.
	DEPC	24 (SEL0)	Contains the address of the instruction that caused a debug exception, from which point processing resumes after the exception has been serviced. Also saves the ISA mode bit that was in effect before the exception occurred.
	DESAVE	31 (SEL0)	Debug exception save register for exclusive use by an in-circuit emulator (ICE).

The sections in this chapter describe the CP0 register organizations and how data is represented in these registers. The number following a register name in the headings as in "8.2.1 Config Register (16:SEL0)" indicates the register number.

8.2 System Configuration Registers

8.2.1 Config Register (16:SEL0)

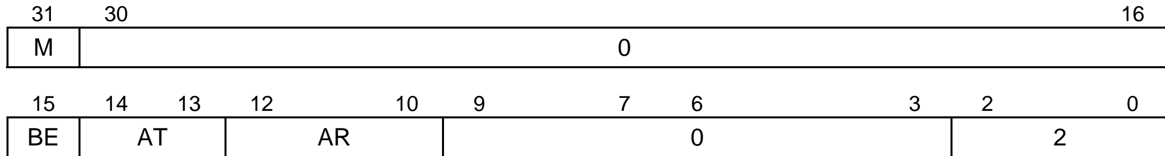


Table 8-2 Config Register Field Descriptions

Fields		Description	Read/ Write	Reset Value
Name	Bits			
M	31	Config1 Register Select Field In the TX19A, this field is fixed at 1.	R	1
–	30:16	Reserved This field is always read as 0.	R	0
BE	15	Endian is selective and fixed at mounting. Endian: 0: Little-endian 1: Big-endian	R	Note 1
AT	14:13	Architecture Type: 0: MIPS32 1: MIPS64 with access only to 32-bit compatibility segments 2: MIPS64 with access to all address segments 3: Reserved In the TX19A, this field is fixed at 0.	R	0
AR	12:10	Architecture Revision Level: 0: Revision 1 1-7: Reserved In the TX19A, this field is fixed at 0.	R	0
–	9:3	In the TX19A, this field is fixed at 0.	R	0
–	2:0	In the TX19A, this field is fixed at 2.	R	2

Note 1: Endian is fixed as 0 or 1 at mounting.

8.2.2 Config1 Register (16:SEL1)

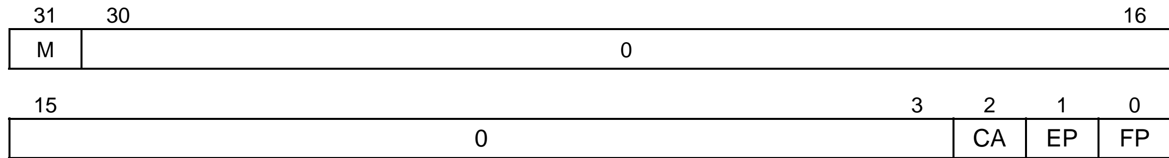


Table 8-3 Config1 Register Field Descriptions (1 of 3)

Fields		Description	Read/ Write	Reset Value
Name	Bits			
M	31	Config2 Register Select Field: In the TX19A, this bit is fixed at 1.	R	1
—	30:3	In the TX19A, this bit is fixed at 0.	R	0
CA	2	16-bit code Implemented. 0: MIPS16ASE not implemented 1: MIPS16ASE implemented In the TX19A, this bit is fixed at 1.	R	1
EP	1	EJTAG Implemented: 0: No EJTAG implemented 1: EJTAG implemented In the TX19A, this bit is fixed at 1.	R	1
FP	0	FPU Implemented: 0: No FPU implemented 1: FPU implemented In the TX19A, this bit is fixed at 0.	R	0

Note: The Config1 register is read-only.

8.2.3 Config2 Register (16:SEL2)

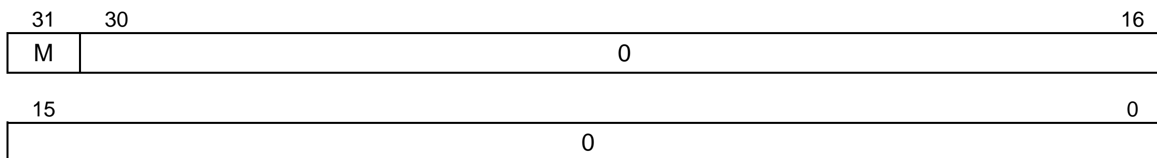


Table 8-4 Config2 Register Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
M	31	Config3 Register Select Field: In the TX19A, this bit is fixed at 0.	R	1
–	30:0	In the TX19A, this bit is fixed at 0.	R	0

Note: The Config2 register is read-only.

8.2.4 Config3 Register (16:SEL3)

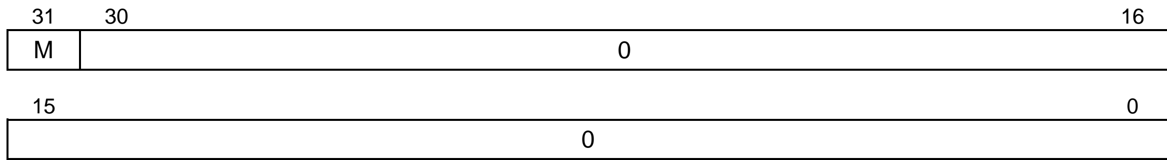


Table 8-5 Config3 Register Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
M	31	Config4 Register Select Field: In the TX19A, this bit is fixed at 0.	R	0
–	30:0	In the TX19A, this bit is fixed at 0.	R	0

Note: The Config3 register is read-only.

8.3 General Exception Handling Registers

This section describes the CP0 registers that are used in general exception processing. The remaining CP0 registers are used for program debug and described in the next section.

8.3.1 BadVAddr Register (8)

The BadVAddr (Bad Virtual Address) register is a read-only register. It captures the most recent virtual address that caused a virtual-to-physical address translation error. The Address Error (AdEL or AdE) exception is taken.

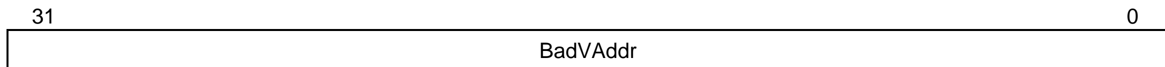


Table 8-6 BadVAddr Register Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
BadVAddr	31:0	Bad Virtual Address	R	Undefined

Note: BadVAddr register is read-only.

8.3.2 Count Register (9:SEL0)

The Count register is a read/write register. It acts as a time, incrementing at 1/2 the rate of CPUCLK.

If the processor input called GTINTDIS is held at logic 0, the Count register is incremented. If GTINTDIS is held at logic 1, the Count register remains inactive.

The Count register can be written for diagnostic purposes or during system initialization.

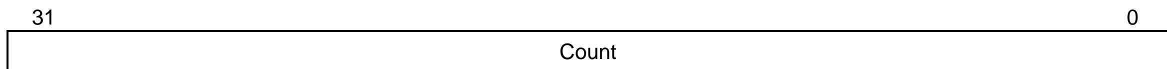


Table 8-7 Count Register Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
Count	31:0	Interval counter	R/W	Undefined

8.3.3 Compare Register (11)

When the value of the Count register reaches the value programmed into the Compare register, interrupt bit IP[7] in the Cause register is set. This causes an Interrupt exception if the interrupt is enabled.

Writing to the Compare register clears the timer interrupt.

For diagnostic purposes, the Compare register is a read/write register. In normal use, the Compare register is write-only.

Table 8-8 Compare Regiser Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
Compare	31:0	Interval count compare value	R/W	Undefined

8.3.4 Status Register (12)

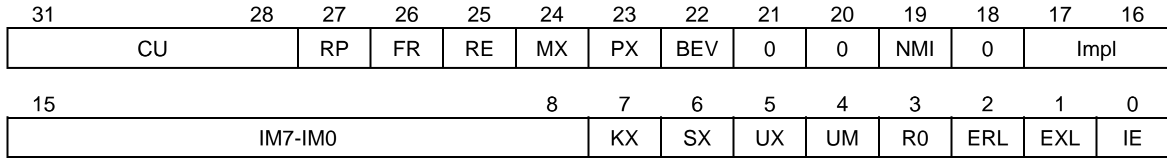


Table 8-9 Status Register Field Descriptions (1 of 2)

Field Name	Field Bits	Description	Read/ Write	Reset Value
CU (CU3, … CU0)	31:28	Controls the usability of coprocessors 3 to 0. In Kernel mode, CP0 is always usable, regardless of the value of the CU0 bit. The CU3, CU2 and CU1 bits must always be written as 0s. 0: Coprocessor not usable 1: Coprocessor usable	R/W	Undefined
RP	27	Reduced Power Mode: 0: Halt mode 1: Doze Mode Selects which one of the reduced power modes is to be entered on execution of an WAIT instruction. The TX19A freezes the instruction pipeline in both Halt and Doze modes; however, the Halt mode provides more power savings than the Doze mode.	R/W	0
FR	26	Ignored on write and returned as 0 on read	R	0
RE	25	Ignored on write and returned as 0 on read	R	0
MX	24	Ignored on write and returned as 0 on read	R	0
PX	23	Ignored on write and returned as 0 on read	R	0
BEV	22	Bootstrap Exception Vector Set when the processor is reset. When BEV=1, all exception vectors reside in uncacheable kseg1 space. Typically, this is used to allow diagnostic tests to occur before the functionality of the cache is validated. When BEV=0, the Reset, NMI and Debug exception vectors reside in uncacheable kseg1 space and all the other exception vectors reside in cacheable kseg0 space.	R/W	1
TS	21	Ignored on write and returned as 0 on read.	R	0
SR	20			
NMI	19	Set when a non-maskable interrupt (NMI) signal is asserted low. Writing a 0 to this bit clears it. Writing a 1 to this bit has no effect. Note that the functionality of this bit differs between the TX19.	R/W	0
—	18	Ignored on write and returned as 0 on read.	R	0
Impl	17:16	Ignored on write and returned as 0 on read.	R	0
IM (IM7, … IM0)	15:8	Interrupt Mask: Enables and disables each of the external, timer and software interrupts. An interrupt is only accepted when the Interrupt Enable (IE) bit is set and the corresponding IM bit in the Status register and the IP bit in the Cause register are both set. 0: Interrupt request disabled 1: Interrupt request enabled	R/W	0x00

Table 8-9 Status Register Field Descriptions (2 of 2)

Field		Description	Read/ Write	Reset Value
Name	Bits			
KX	7	Ignored on write and returned as 0 on read.	R	0
SX	6	Ignored on write and returned as 0 on read.	R	0
UX	5	Ignored on write and returned as 0 on read.	R	0
UM	4	Operating Mode: 0: Kernel mode 1: User mode Only Kernel mode is available with TX19A.	R/W	0
—	3	Ignored on write and returned as 0 on read.	R	0
ERL	2	Error Level: Set when a Reset or NMI exception is taken. When this bit is set: • The processor is running is Kernel mode. • Interrupts are disabled. • The ERET instruction will use the return address held in the ErrorEPC register.	R/W	1
EXL	1	Exception Level: Set when an exception other than Reset and NMI exceptions is taken. When this bit is set: • The processor is running in Kernel mode. • Interrupts are disabled. • The EPC register and the BD bit in the Cause register will not be updated if another exception is taken.	R/W	0
IE	0	Interrupt Enable: 0: Interrupts are disabled. 1: Interrupts are enabled. The IE bit is not automatically set or cleared by the interrupt response sequence or the ERET instruction. (This bit is cleared upon reset.)	R/W	0

8.3.5 Cause Register (13)

The Cause register indicates the cause of the last exception. All the bits in this register is read-only, except the IP[1:0] and IV bits.

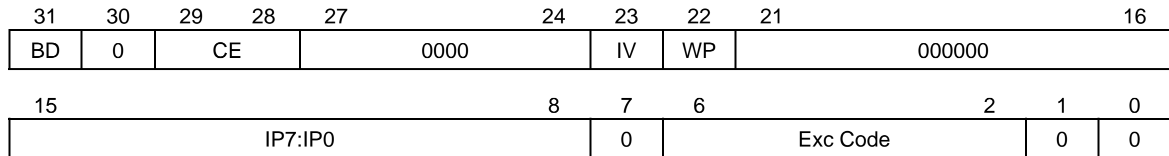


Table 8-10 Cause Register Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
BD	31	Set when an exception occurred in a jump or branch delay slot. The processor updates the BD bit only if the EXL bit is 0 when an interrupt or exception occurred.	R	Undefined
–	30	Ignored on write and returned as 0 on read.	R	0
CE[1:0]	29:28	Coprocessor Error: Indicates the coprocessor unit number referenced when a Coprocessor Unusable exception was taken. The value in this field is undefined for any other exception.	R	Undefined
–	27:24	Ignored on write and returned as 0 on read.	R	0
IV	23	Interrupt Vector: If this bit is set, an Interrupt exception uses a special interrupt vector different from the general exception vector. BEV (Status) IV Interrupt Vector 0 0 0x8000_0180 0 1 0x8000_0200 1 0 0xBFC0_0380 1 1 0xBFC0_0400	R/W	Undefined
WP	22	Ignored on write and returned as 0 on read.	R	0
–	21:16	Ignored on write and returned as 0 on read.	R	0
IP[7:2]	15:10	Interrupt Request (Hardware): IP[7]: Hardware interrupt 5 or timer interrupt IP[6]: Hardware interrupt 4 IP[5]: Hardware interrupt 3 IP[4]: Hardware interrupt 2 IP[3]: Hardware interrupt 1 IP[2]: Hardware interrupt 0 A timer interrupt occurs when the value of the Count register ($9) equals the value of the Compare register ($11).	R	Undefined
IP[1:0]	9:8	Interrupt Request (Software) IP[1]: Request software interrupt 1 IP[0]: Request software interrupt 0	R/W	Undefined
–	7	Ignored on write and returned as 0 on read.	R	0
ExcCode	6:2	Exception code (See Table 8-11.)	R	Undefined
–	1:0	Ignored on write and returned as 0 on read.	R	0

Table 8-11 Exception Code (ExcCode) Field

Exception Code Value		Mnemonic	Description
Decimal	Hexadecimal		
0	0x00	Int	Interrupt exception (software and hardware)
4	0x04	AdEL	Address Error exception (instruction fetch or load)
5	0x05	AdES	Address Error exception (Store)
6	0x06	IBE	Bus Error exception (instruction fetch)
7	0x07	DBE	Bus Error exception (data load)
8	0x08	Sys	System Call exception
9	0x09	Bp	Breakpoint exception
10	0x0a	RI	Reserved Instruction exception
11	0x0b	CpU	Coprocessor Unusable exception
12	0x0c	Ov	Integer Overflow exception
13	0x0d	Tr	Trap exception
Other	(Reserved)		

8.3.6 EPC Register (14)

The EPC register contains the address at which processing resumes after an exception has been serviced.

For synchronous exceptions, the EPC register contains either one of the following:

- the virtual address of the instruction that was the direct cause of the exception
- the virtual address of the immediately preceding branch or jump instruction (When the exception-causing instruction is in a branch delay slot, the BD bit in the Cause register is set.)

The processor does not write to the EPC register when the EXL bit in the Status register is set to one.

Table 8-12 EPC Register Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
EPC	31:0	Exception Program Counter	R/W	Undefined

8.3.7 PRId Register (15)

The PRId register is a read-only register that indicates the implementation and revision identifier of the CPU and the CP0.

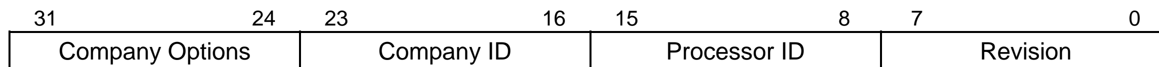


Table 8-13 PRId Register Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
Company Options	31:24	Company-dependent options Returned as 0 on read.	R	0x00
Company ID	23:16	Company ID In the TX19A, this field is fixed at 0x07.	R	0x07
Processor ID	15:8	Processor ID In the TX19A, this field is fixed at 0x40.	R	0x40
Revision	7:0	Revision In the TX19A, this field is fixed at 0x00.	R	0x00

Note: PRId register is read-only.

8.3.8 ErrorEPC Register (30)

The ErrorEPC register is a read/write register that captures the value of the Program Counter (PC) on Reset and NMI exceptions.

The ErrorEPC register contains one of the following addresses:

- the virtual address of the instruction that was the direct cause of the exception
- the virtual address of the immediately preceding branch or jump instruction when the error-causing instruction is in a branch delay slot

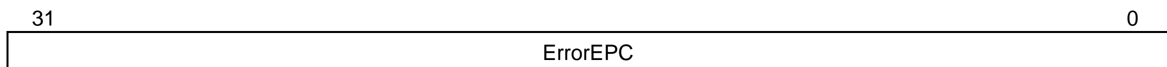


Table 8-14 ErrorEPC Register Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
ErrorEPC	31:0	Error Exception Program Counter	R/W	Undefined

8.3.9 Shadow Register Set Control Register: SSCR (22 or 9：SEL6)

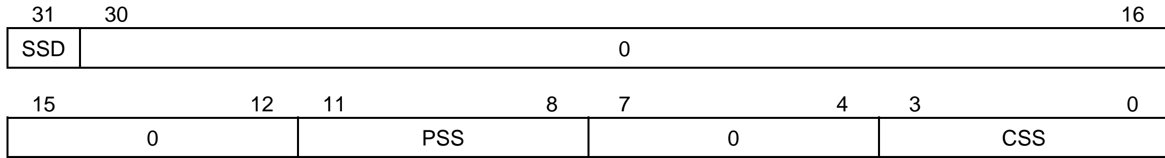


Table 8-15 SSCR Register Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
SSD	31	Shadow Register Set Disable Signal 0: MIPS32 Version.1.0 with Shadow Register Set 1: MIPS32 Version.1.0	R/W	1
–	30:12	Reserved bits	R	0
PSS	11:8	Previous Shadow Register Set x000: Main GPRs x001: Shadow Register 1 x010: Shadow Register 2 x011: Shadow Register 3 x100: Shadow Register 4 x101: Shadow Register 5 x110: Shadow Register 6 x111: Shadow Register 7	R/W	Undefined
–	7:4	Reserved bits	R	0
CSS	3:0	Current Shadow Register Set x000: Main GPRs x001: Shadow Register 1 x010: Shadow Register 2 x011: Shadow Register 3 x100: Shadow Register 4 x101: Shadow Register 5 x110: Shadow Register 6 x111: Shadow Register 7	R/W	0000

Note 1: The SSCR register is a read/write register.

Note 2: When the processor accepts an interrupt request from the interrupt controller, the value of the CSS field is copied to the PSS field, and the CSS field is updated with the value of the new interrupt request level.

Note 3: On an ERET, the value of the PSS field is restored to the CSS field.

Note 4: The instruction that modifies the contents of the SSCR register must be followed by two NOPs to avoid pipeline hazards.

Example:
```
MTC0    r18, SSCR
NOP
NOP
ADD     r19, r12, r13
```

Note 5: When the SSD bit is set, the Shadow Register Set is not updated by any interruptions.

Note 6: When the SSD bit is set, only shadow set 0 is accessible, and the value of the CSS field is ignored.

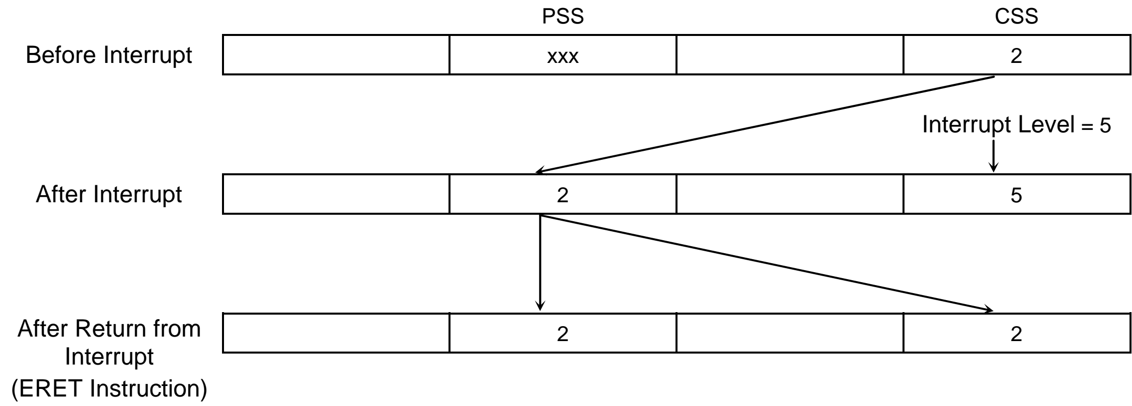


Figure 8-16 Saving and Restoring the Shadow Register Set Number

8.3.10 IER Register (9:SEL7)

The IER register is used to set or clear the IE bit in the Status register. Writing a zero to the IER register causes the IE bit in the Status register to be cleared. Writing a non-zero value to the IER register causes the IE bit to be set. Use the instruction “MTC0 r0, IER” to disable interrupts. Use a register that contains a non-zero value as the target register like “MTC0 $sp, IER” to enable interrupts.

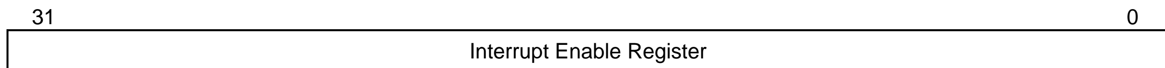


Table 8-17 IER Register Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
IER	31:0	Interrupt Enable Register This register is used to set or clear the IE bit in the Status register. Writing a 0 to this register causes the IE bit to be cleared. Writing a non-zero value to this register causes the IE bit to be set.	R/W	Undefined

8.4 Debug Exception Handling Registers

The TX19A allows program instruction execution to arbitrarily stop to handle debugging events.
This section provides explanations about the extra hardware-based features the TX19A incorporate to enhance program debug.

8.4.1 Debug Register (23)

As a debugging aid, the Debug register reflects conditions that were in effect at the time a debug exception occurred and allows you to initiate debug processing. Code execution breakpoints can be generated by embedding Software Debug Breakpoint (SDBBP) instructions in the code at the time SDBBP instruction is executed.
Additionally, the single-step feature may be enabled by setting the SSt bit in the Debug register; when single-step mode is enabled, a Single-Step exception occurs each time the processor executes an instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DBD	DM	No DCR	LSNM	Doze	Halt	Count DM	IBus EP	M CheckP`	Cache EP	DBus EP	IEXI	DDBS Impr	DDBL Impr	EJTAG ver[2:0]	

15	14				10	9	8	7	6	5	4	3	2	1	0
EJTAG ver[2:0]	DexcCode					NoSSt	SSt	0		DINT	DIB	DDBS	DDBL	DBp	DSS

Table 8-18 Debug Register Field Descriptions (1 of 3)

Field		Description	Read/ Write	Reset Value
Name	Bits			
DBD	31	Debug Branch Delay: Set when a debug exception occurred in a jump or branch delay slot.	R	Undefined
DM	30	Debug Mode: Indicates whether a debug exception occurred. This bit is set when a debug exception is taken and cleared on a DERET.	R	0
NoDCR	29	dseg memory segment: 0: Dseg is present 1: No dseg present	R	0
LSNM	28	Controls access of load/store between dseg and remaining memory when dseg is present: 0: Load/store in dseg address range go to dseg 1: Load/store in dseg address range go to system memory	R	0

Table 8-18 Debug Register Field Descriptions (2 of 3)

Field		Description	Read/ Write	Reset Value
Name	Bits			
Doze	27	Low-Power Mode Flag (Doze): Set if the processor was in low-power Doze mode when a debug exception occurred.	R	Undefined
Halt	26	Low-Power Mode Flag (Halt): Set if the processor was in low-power Halt mode when a debug exception occurred.	R	Undefined
CountDM	25	Count Register in Debug Mode: 0: Stops the Count register in Debug mode. 1: Runs the Count register in Debug mode.	R/W	0
IBusEP	24	Instruction Bus Error Pending: Set when a bus error (from an instruction fetch) is detected or a 1 is written to the bit by software in debug mode. Cleared when a Bus Error exception is taken by the processor. If the IEXI bit is cleared when the IBusEP bit is set, the pending Bus Error exception is taken by the processor, and the IBusEP bit is cleared. Writing a 0 to this bit has no effect.	R/W1	0
McheckP	23	Not implemented in the TX19A. Returned as 0 on read.	R	0
CacheEP	22	Not implemented in the TX19A. Returned as 0 on read.	R	0
DBusEP	21	Data Bus Error Pending: Set when a bus error (from a data access) is detected or a 1 is written to the bit by software in debug mode. Cleared when a Bus Error Exception is taken by the processor. If the IEXI bit is cleared when the DBusEP bit is set, the pending Bus Error exception is taken by the processor, and the DBusEP bit is cleared. Writing a 0 to this bit has no effect.	R/W1	0
IEXI	20	An Imprecise Error eXception Inhibit (IEXI) Set when the processor takes a debug exception or an exception occurs in Debug mode. Cleared by the DERET instruction. Also modifiable by software. When the IEXI bit is set, Bus Error exceptions (from instruction fetches and data accesses) are inhibited or deferred until the bit is cleared.	R/W	0
DDBS Impr	19	Debug Data Break Store Imprecise Exception: Set when a data address break occurred during a write bus cycle. Cleared when a general exception occurred in Debug mode.	R	Undefined
DDBL Impr	18	Debug Data Break Load Imprecise Exception: Set when a data address break occurred during a read bus cycle. Cleared when a general exception occurred in Debug mode.	R	Undefined
EJATGver	17:15	EJTAG version: 0: Version 1 and 2.0 1: Version 2.5 2: Version 2.6 3-7: Reserved	R	010
DExcCode	14:10	General Exception in Debug Mode: Indicates the Exception Code (ExcCode) in the same manner as for the Cause register if a general exception occurs while a debug exception handler is being executed in Debug mode (i.e., DM=1). See Table 8-11 for a list of exception codes.	R	Undefined
NoSSt	9	Single-Step Feature Available: 0: Single-step feature available 1: No single-step feature available In the TX19A, this bit is fixed at 0.	R	0

Table 8-18 Debug Register Field Descriptions (3 of 3)

Field		Description	Read/ Write	Reset Value
Name	Bits			
SSt	8	Single-Step: When set, the single-step feature is enabled. When cleared, the single-step feature is disabled. The single-step feature is disabled while a debug exception handler is being executed (i.e., DM=1).	R/W	0
0	7:6	Ignored on write and returned as 0 on read.	0	0
DINT	5	Debug Interrupt Exception: Set when a Debug Interrupt exception occurred. Cleared on a general exception in Debug mode.	R	Undefined
DIB	4	Debug Instruction Break: Set when an instruction address break occurred. Cleared on a general exception in Debug mode.	R	Undefined
DDBS	3	Debug Data Break Store Exception: Debug Data Break Store Exception: Set when a data address break occurred on a store. Cleared on a general exception in Debug mode. The Debug Data Break Store exception is not implemented in the TX19A.	R	Undefined
DDBL	2	Debug Data Break Load Exception: Set when a data address break occurred on a load. The breakpoint match is evaluated on a load address, but not on the data value. Cleared on a general exception in Debug mode. The Debug Data Break Load exception is not implemented in the TX19A.	R	Undefined
DBp	1	Debug Breakpoint Exception: Set when an SDBBP instruction caused a Debug Breakpoint exception. Cleared on a general exception in Debug mode.	R	Undefined
DSS	0	Debug Single-Step Exception: Set when a Single-Step exception occurred. Cleared on a general exception in Debug mode.	R	Undefined

8.4.2 DEPC Register (24)

The DEPC register contains the address at which processing resumes after a debug exception has been serviced.

The DEPC register contains either one of the following:

- the virtual address of the instruction that was the direct cause of the debug exception.
- the virtual address of the immediately preceding branch or jump instruction when the exception causing instruction is in a branch delay slot

The DERET instruction causes a jump to DEPC address. The DEPC register is a read/write register.

Table 8-19 DEPC Register Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
DEPC	31:0	Debug Exception Program Counter	R/W	Undefined

8.4.3 DESAVE Register (31)

The debug exception handler uses the DESAVE register to save one of the general-purpose registers. The general-purpose register saved in DESAVE is used to save the rest of the context to a predetermined memory area, for example, in a processor probe. The DESAVE register allows the safe debugging of exception handlers and other types of code where the existence of a valid stack for context saving can not be assumed.

The DESAVE register is for exclusive use by an in-circuit emulator (ICE).

Table 8-20 DESAVE Register Field Descriptions

Field		Description	Read/ Write	Reset Value
Name	Bits			
DESAVE	31:0	Debug Exception Save Register	R/W	Undefined

TX
System
RISC

Chapter 9 Exception Handling

This chapter discusses system resources related to exception and exception processing sequence. The main sections in this chapter are:

- General Exceptions
- Interrupts
- Debug Exceptions

9.1 General Exceptions

Exceptions in the TX19A are broadly categorized into general exceptions or debug exceptions. This section explains details concerning sources of specific exceptions, how each arises and how each is processed.

9.1.1 How General Exception Processing Works

Exceptions are any conditions that alter the normal sequence of instructions as a result of external interrupt signals, errors or unusual conditions arising in the execution of instructions. When exceptions occur, the processor saves information about the state of the processor, enters Kernel mode and transfers control to a predefined address. This predefined location is called exception vector, which directly indicates the start of the actual exception handler routine.

All exceptions other than Reset and NMI exceptions are processed in the sequence shown in Figure 9-1. Reset and NMI exceptions are processed in the sequence shown in Figure 9-2.

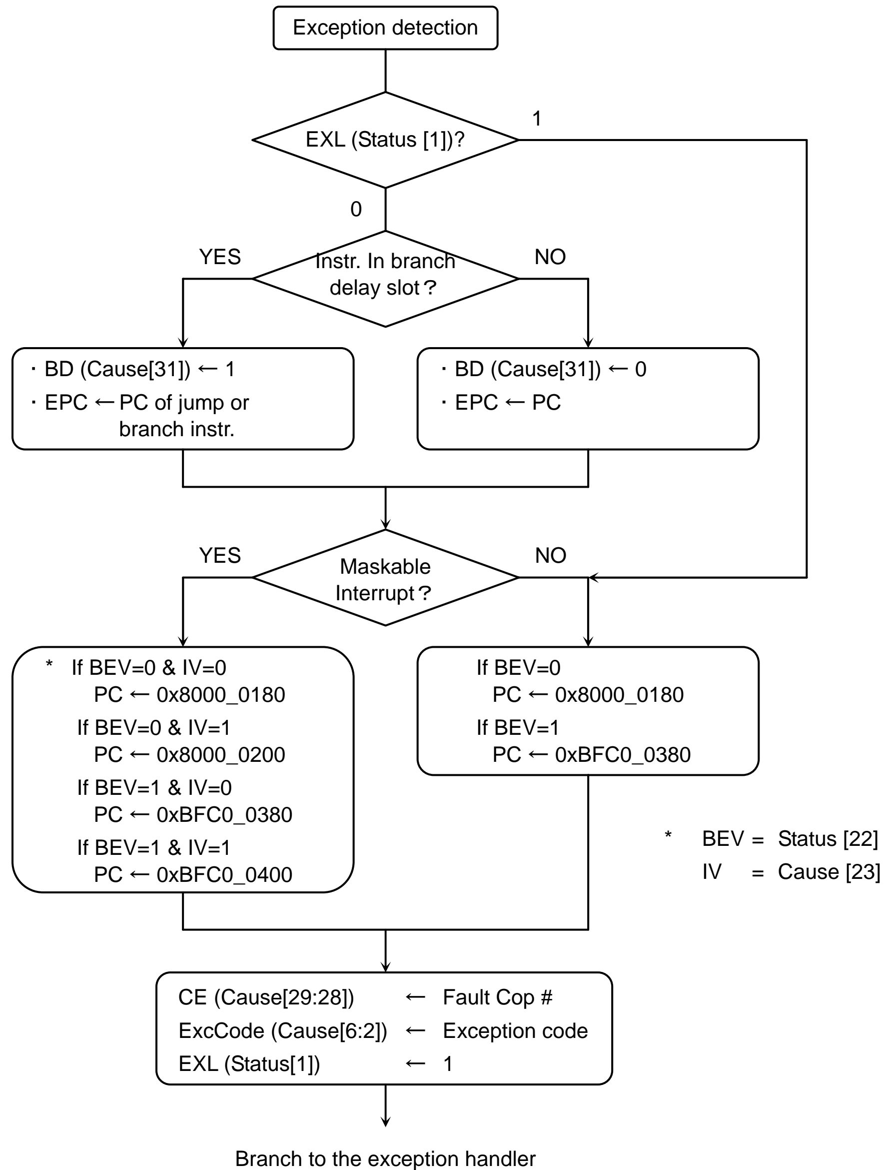


Figure 9-1 General Exception Processing

The CE field consists of the Cause register bit 28 and 29 is only valid when a Coprocessor Unusable exception occurred.

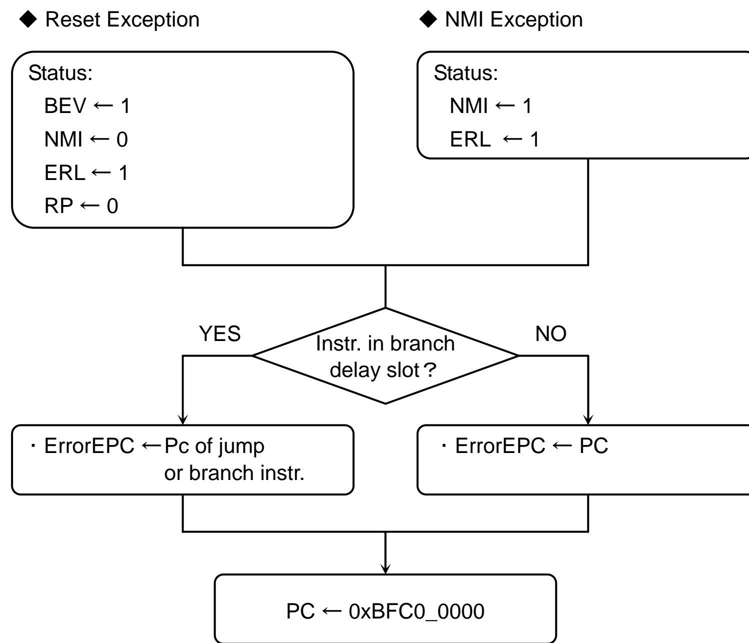


Figure 9-2 Reset and NMI Exception Processing

9.1.2 General Exception Priorities

While more than one exception can occur at a time, the TX19A reports only one exception with the priority order shown in Table 9-3.

Table 9-3 General Exception Types

Priority	Exception	Mnemonic	Type
Highest	Reset	Reset	Non_debug
	Single-Step exception	DSS	Debug
	Nonmaskable Interrupt (NMI) exception	Nmi	Non_debug
	Maskable Interrupt exception	Int	
	Address Error exception (Instruction fetch)	AdEL	Non_debug
	Bus Error exception (Instruction fetch)	IBE	
	Debug Breakpoint exception (SDBBP)	DBp	Debug
	Coprocessor Unusable exception (see Note)	CpU	Non_debug
	Reserved Instruction exception, Integer Overflow exception, Trap exception, System Call exception, Breakpoint exception	RI、Ov、Tr、Sys、Bp	
	Address Error exception (Load/store)	AdEL/AdES	Non_debug
Lowest	Bus Error exception (Data access)	DBE	

Note: When FPU instructions with the COP1 opcode are executed with the CU1 bit cleared, both CpU and RI exception conditions arise. The CpU exception occurs, however, based on the priority order shown above.

9.1.3 Exception Vector Addresses (Exception Vectors)

An exception vector is the entry address of a routine that handles an exception. The Reset and Nonmaskable Interrupt exceptions are always vectored to virtual address 0xBFC0_0000. The Debug exception is always vectored to virtual address 0xBFC0_0480. Values of the other vectors depend on the BEV bit (bit 23) in the Status register and the IV bit (bit 23) in the Cause register. Table 9-4 shows the exception vector addresses.

Table 9-4 Exception Vector Addresses

Exception Type	BEV=0		BEV=1	
	Virtual	**Physical**	**Virtual**	**Physical**
Reset、NMI	0xBFC0_0000	0x1FC0_0000	0xBFC0_0000	0x1FC0_0000
Debug Breakpoint	0xBFC0_0480	0x1FC0_0480	0xBFC0_0480	0x1FC0_0480
Interrupt (IV=0)	0x8000_0180	0x0000_0180	0xBFC0_0380	0x1FC0_0380
Interrupt (IV=1)	0x8000_0200	0x0000_0200	0xBFC0_0400	0x1FC0_0400
All others	0x8000_0180	0x0000_0180	0xBFC0_0380	0x1FC0_0380

9.1.4 Reset Exception

■ Cause

This exception occurs when the processor's reset signal is asserted and then negated.

■ Handling

1. All the CP0 registers are initialized.
2. The ERL bit in the Status register is set.
3. The ErrorEPC register is loaded with the restart PC.
4. The processor jumps to the exception handler located at address 0xBFC0_0000.

Note: If a Reset exception occurs during processor bus cycles, the processor immediately discontinues the ongoing bus cycle and takes a Reset exception.

9.1.5 Nonmaskable Interrupt (NMI) Exception

■ Cause

This exception occurs when the processor's non-maskable interrupt signal, GNMI, is asserted. This exception is not maskable; it occurs regardless of the settings of the EXL, ERL and IE bits in the Status register.

■ Handling

1. The values of the ExcCode and CE fields in the Cause register become undefined.
2. The ERL and NMI bits in the Status register are set.
3. The ErrorEPC register is loaded with the program counter (PC) on the interrupt. If the interrupt-causing instruction is in a jump or branch delay slot, the ErrorEPC register points at the preceding jump or branch instruction, and the BD bit in the Cause register is set. The least-significant bit in the ErrorEPC register saves the ISA mode that was in effect prior to the exception.
4. If the processor is in a reduced power mode (either HALT or DOZE mode), the reduced power mode is exited and start to handle the NMI exception.
5. If the exception occurs while the processor is in 16-bit ISA mode, the processor switches to 32-bit ISA mode.
6. The processor jumps to the exception handler located at address 0xBFC0_0000.

Note: When an NMI interrupt request is generated during a bus cycle, the processor recognizes the request at the end of the current bus cycle, as is the case with all the other exceptions but the Reset exception.

9.1.6 Address Error Exception

■ Cause

This exception occurs when an attempt is made to:

- fetch a 32-bit ISA instruction that is not aligned on a word boundary (AdEL)
- fetch a 16-bit ISA instruction that is not aligned on a halfword boundary (AdEL)
- load or store a word that is not aligned on a word boundary (AdEL/AdES)
- load or store a halfword that is not aligned on a halfword boundary (AdEL/AdES)
- reference a Kernel-mode address space (kseg0, kseg1 or kseg2) in User mode (AdEL/AdES)

■ Handling

1. The AdEL code (4) or the AdES code (5) is set into the ExcCode field in the Cause register, depending on whether the exception occurred during an instruction fetch or a load operation (AdEL), or a store operation (AdES).
2. The BadVAddr register stores the virtual address that is not properly aligned or the virtual address that improperly references a Kernel segment.
3. The following operation only occurs when the EXL bit in the Status register is cleared. The EXL bit is set, and the EPC register is loaded with the address of the instruction that caused the exception unless this instruction is not in a jump or branch delay slot. If it is in a jump or branch delay slot, the EPC register points at the preceding jump or branch instruction and the BD bit in the Cause register is set. The least-significant bit of the EPC register saves the ISA mode that was in effect prior to the exception.
4. If the exception occurs while the processor is in 16-bit ISA mode, the processor switches to 32-bit ISA mode.
5. The processor jumps to an appropriate exception vector address (see Table 9-5).

9.1.7 Bus Error Exception

■ Cause

This exception occurs when the bus error signal, GBUSERR, is asserted during memory read bus cycles. A Bus Error exception can occur during the fetching of any instruction or during a memory read bus cycle by a load or bit manipulation instruction.

The handling of a write bus error differs between the TX19 and the TX19A. The assertion of the GBUSERR signal causes the TX19 to take a Bus Error exception whether or not it is during a read or write operation. In the TX19A, GBUSERR is not signaled to the processor during a write operation because of the on-chip write buffer; in case of a write bus error, the system hardware must terminate the write operation through use of an NMI interrupt.

■ Handling

1. The IBE code (6) or the DBE code (7) is set into the ExcCode field in the Cause register, depending on whether the exception occurred during an instruction fetch (IBE), or a data load or store operation (DBE).
2. The following operation only occurs when the EXL bit in the Status register is cleared. The EXL bit is set, and the EPC register is loaded with the address of the instruction that caused the exception unless this instruction is not in a jump or branch delay slot. If it is in a jump or branch delay slot, the EPC register points at the preceding jump or branch instruction and the BD bit in the Cause register is set. The least-significant bit of the EPC register saves the ISA mode that was in effect prior to the exception.
3. The EPC register saves the program counter (PC) on the exception for the following cases:

- a load instruction is followed by a SYNC instruction
- the instruction immediately following a load has dependency on the loaded data

In such cases, the pipeline stalls until the load is complete; so the EPC register displays the address of the instruction immediately following the load instruction.

4. If the exception occurs while the processor is in 16-bit ISA mode, the processor switches to 32-bit ISA mode.

5. The processor jumps to an appropriate exception vector address (see Table 9-4).

9.1.8 Integer Overflow Exception

■ Cause

This exception occurs when the ADD, ADDI or SUB instruction in the 32-bit ISA or the DIVE instruction in the 16-bit ISA results in two's-complement overflow or when the DIVE or DIVEU instruction in the 16-bit ISA attempts to divide by zero.

■ Handling

1. The Ov code (12) is set into the ExcCode field in the Cause register.
2. The following operation only occurs when the EXL bit in the Status register is cleared. The EXL bit is set, and the EPC register is loaded with the address of the instruction that caused the exception unless this instruction is not in a jump or branch delay slot. If it is in a jump or branch delay slot, the EPC register points at the preceding jump or branch instruction and the BD bit in the Cause register is set. The least-significant bit of the EPC register saves the ISA mode that was in effect prior to the exception.
3. If the exception occurs while the processor is in 16-bit ISA mode, the processor switches to 32-bit ISA mode.
4. The processor jumps to an appropriate exception vector address (see Table 9-4).

9.1.9 Trap Exception

■ Cause

This exception occurs when the TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEIU, TLTI, TLTIU, TEQI or TNEI instruction results in a true condition.

■ Handling

The Tr code (13) is set into the ExcCode field in the Cause register.

1. The following operation only occurs when the EXL bit in the Status register is cleared.
2. The EXL bit is set, and the EPC register is loaded with the address of the instruction that caused the exception unless this instruction is not in a jump or branch delay slot. If it is in a jump or branch delay slot, the EPC register points at the preceding jump or branch instruction and the BD bit in the Cause register is set. The least-significant bit of the EPC register saves the ISA mode that was in effect prior to the exception.
3. The processor jumps to the appropriate exception vector address (see Table 9-4).

* The Trap exception occurs only in 32-bit ISA mode.

9.1.10 System Call Exception

■ Cause

This exception occurs when a SYSCALL instruction is executed.

■ Handling

1. The Sys code (8) is set into the ExcCode field in the Cause register.

2. The following operation only occurs when the EXL bit in the Status register is cleared. The EXL bit is set, and the EPC register is loaded with the address of the instruction that caused the exception unless this instruction is not in a jump or branch delay slot. If it is in a jump or branch delay slot, the EPC register points at the preceding jump or branch instruction and the BD bit in the Cause register is set. The least-significant bit of the EPC register saves the ISA mode that was in effect prior to the exception.

3. If the exception occurs while the processor is in 16-bit ISA mode, the processor switches to 32-bit ISA mode.
4. The processor jumps to an appropriate exception vector address (see Table 9-4).

When a System Call exception occurs, control is transferred to an exception handler. The unused bits (bits 25-6 in the 32-bit ISA; bits 25-16 and 10-5 in the 16-bit ISA) in a SYSCALL instruction is available for use as software parameters to pass additional information. To examine these bits, load the contents of the instruction at which the EPC register points. If the instruction is in a jump or branch delay slot (i.e., the BD bit in the Cause register is set), add four to the contents of the EPC register to locate the instruction.

To resume execution after the exception has been serviced, alter the contents of the EPC register by adding four so that the SYSCALL instruction is not re-executed. If the SYSCALL instruction is in a jump or branch delay slot (i.e., the BD bit in the Cause register is set), the instruction at the return address is a jump or branch instruction. In that case, the jump or branch instruction must be interpreted to set the EPC register before resuming execution.

9.1.11 Breakpoint Exception

■ Cause

This exception occurs when a BREAK instruction is executed.

■ Handling

1. The Bp code (9) is set into the ExcCode field in the Cause register.
2. The following operation only occurs when the EXL bit in the Status register is cleared. The EXL bit is set, and the EPC register is loaded with the address of the instruction that caused the exception unless this instruction is not in a jump or branch delay slot. If it is in a jump or branch delay slot, the EPC register points at the preceding jump or branch instruction and the BD bit in the Cause register is set. The least-significant bit of the EPC register saves the ISA mode that was in effect prior to the exception.
3. If the exception occurs while the processor is in 16-bit ISA mode, the processor switches to 32-bit ISA mode.
4. The processor jumps to the appropriate exception vector address (see Table 9-4).

When a Breakpoint exception occurs, control is transferred to an exception handler. The unused bits (bits 25-16 in the 32-bit ISA, bits 10-5 in the 16-bit ISA) in a BREAK instruction is available for use as software parameters to pass additional information. To examine these bits, load the contents of the instruction at which the EPC register points. If the instruction is in a jump or branch delay slot (i.e., the BD bit in the Cause register is set), add four (in the 32-bit ISA mode) or two (in the 16-bit ISA mode) to the contents of the EPC register to locate the instruction.

To resume execution after the exception has been serviced, alter the contents of the EPC register by adding four (in 32-bit ISA mode) or two (in 16-bit ISA mode) so that the BREAK instruction is not re-executed. If the BREAK instruction is in a jump or branch delay slot (i.e., the BD bit in the Cause register is set), the instruction at the return address is a jump or branch instruction. In that case, the jump or branch instruction must be interpreted to set the EPC register before resuming execution.

9.1.12 Reserved Instruction Exception

■ Cause

In 32-bit ISA mode, this exception occurs when an attempt is made to:

- execute an instruction with an undefined major opcode (bits 31-26)
- execute a SPECIAL instruction with an undefined minor opcode (bits 5-0)
- execute a SPECIAL2 instruction with an undefined minor opcode (bits 5-0)
- execute a REGIMM instruction with an undefined minor opcode (bits 20-16)
- execute a COP*z* *rs* instruction (*z*=1 or 2) with an undefined minor opcode (bits 25-21)
- execute a LWC*z*, SWC*z*, LDC*z*, SDC*z* (*z*=1 or 2) or MOVCI instruction

In 16-bit ISA mode, this exception occurs when an attempt is made to:

- execute an instruction with an undefined instruction code: 11101xxxxxx01001, 11101xxxxxx10011, 11101xxxx1100000, 11101xxx01010001, 11101xxx01110001, 11101xxx11010001 or 11101xxx11110001
- execute an unimplemented instruction (LWU, LD, SD, DADDU, DSUBU, DADDIU, DMULT, DMULTU, DDIV, DDIVU, DSLL, DSRL, DSRA, DSLLV, DSRLV, DSRAV)
- EXTEND an instruction that is not extensible
- execute an instruction with an undefined EXTEND+RR minor opcode (bits 4-0)
- execute an instruction with an undefined EXTEND+ADDIU8 minor opcode (bits 7-5 = 001 or 011)
- execute an instruction with an undefined EXTEND+INT minor opcode ([7][1:0] = 100 & [10:8] ≠ 00x)

Handling

1. The RI code (10) is set into the ExcCode field in the Cause register.
2. The following operation only occurs when the EXL bit in the Status register is cleared. The EXL bit is set, and the EPC register is loaded with the address of the instruction that caused the exception unless this instruction is not in a jump or branch delay slot. If it is in a jump or branch delay slot, the EPC register points at the preceding jump or branch instruction and the BD bit in the Cause register is set. The least-significant bit of the EPC register saves the ISA mode that was in effect prior to the exception.
3. If the exception occurs while the processor was in 16-bit ISA mode, the processor switches to 32-bit ISA mode.
4. The processor jumps to an appropriate exception vector address (see Table 9-4).

9.1.13 Coprocessor Unusable Exception

■ Cause

This exception occurs when an attempt is made to:

- execute a CP0 instruction in User mode when the CU0 bit in the Status register is cleared (Kernel- and Debug-mode execution of CP0 instructions never causes this exception, regardless of the setting of the CU0 bit)
- execute COP1, LWC1, SWC1, LDC1, SDC1 or MOVCI instruction when the CU1 bit in the Status register is cleared
- execute COP2, LWC2, SWC2, LDC2 or SDC2 instruction when the CU2 bit in the Status register is cleared
- execute COP3 instruction when the CU3 bit in the Status register is cleared

■ Handling

The CpU code (11) is set into the ExcCode field in the Cause register.

1. The CE field in the Cause register shows which of the coprocessor units was referenced when an exception occurred.
2. The following operation only occurs when the EXL bit in the Status register is cleared. The EXL bit is set, and the EPC register is loaded with the address of the instruction that caused the exception unless this instruction is not in a jump or branch delay slot. If it is in a jump or branch delay slot, the EPC register points at the preceding jump or branch instruction and the BD bit in the Cause register is set. The least-significant bit of the EPC register saves the ISA mode that was in effect prior to the exception.
3. If the exception occurs while the processor is in 16-bit ISA mode, the processor switches to 32-bit ISA mode.
4. The processor jumps to an appropriate exception vector address (see Table 9-4).

9.1.14 Maskable Interrupt Exception (Interrupts)

■ Cause

The TX19A supports the following maskable interrupts:

- Two software interrupts (IP0 and IP1)
- Six hardware interrupts (IP2 to IP7)
- One timer interrupt (IP7)

This exception occurs when all of the following conditions are met:

1. An interrupt request bit in the IP [7:0] field of the Cause register is set (Cause).
2. The corresponding interrupt mask bit in the IM [7:0] field of the Status register is set (Status).
3. The Interrupt Enable (IE) bit in the Status register is set (Status).
4. The processor is not in Debug mode; i.e., the DM bit in the Debug register is cleared (Debug).
5. The Error Level (ERL) and Exception Level (EXL) bits in the Status register are cleared (Status).

An interrupt is taken when all of these conditions are true and a higher-priority exception is not being serviced.

IP7 can be configured for either a hardware interrupt (GINT [5] input) or an internal timer interrupt. The timer interrupt is valid when the GTINTDIS input is at logic 0, and GINT [5] is valid when it is at logic 1.

■ Handling

The interrupt vector address varies, depending on the settings of the BEV bit in the Status register and the IV bit in the Cause register.

Table 9-5 Maskable Interrupt Vectors

IV (Cause[23])	BEV (Status[22])	
	BEV=0	BEV=1
IV=0	0x8000_0180	0xBFC0_0380
IV=1	0x8000_0200	0xBFC0_0400

■ **Servicing**

A software interrupt can be cleared by writing a 0 to the corresponding IP bit (IP1 or IP0) in the Cause register.
A hardware interrupt can be cleared by clearing the cause of the interrupt.
A timer interrupt can be cleared by altering the Compare register value.

9.2 Interrupts

The TX19A provides a non-maskable interrupt and maskable hardware and software interrupts. This section describes the types of interrupts, how interrupts are prioritized and how interrupts are recognized by the processor.

9.2.1 Interrupt Types

The TX19A recognizes a non-maskable interrupt, six maskable hardware interrupts and two maskable software interrupts. Interrupt exceptions are processed by hardware and then serviced by software (interrupt service routines). See 9.1.14, *Maskable Interrupt Exception* and 9.1.5, *Nonmaskable Interrupt (NMI) Exception* for how interrupt exceptions are handled by processor hardware.

Sources of non-maskable interrupts can be an assertion of the processor’s input or on-chip peripherals such as watchdog timers. See individual hardware user’s manuals for possible on-chip sources of non-maskable interrupts. Non-maskable interrupts are for implementation of critical interrupt routines and can not be masked (disabled) by software; they are always recognized regardless of CPU operation mode and forces the processor to restart at 0xBFC0_0000.

Maskable hardware interrupts are detected with the processor’s 3-bit interrupt port. Interrupt requests originate from external or on-chip hardware resources. Interrupt requests are submitted to the interrupt controller, which then turns them into a 3-bit priority level. The priority-level signals are connected to the IP4, IP3 and IP2 ports of the TX19A processor core. The TX19A automatically switches to a specific shadow register set associated with the conditions of IP4, IP3 and IP2 immediately after the interrupt receipt. Thus, for the processor to accept a maskable hardware interrupt, the IM [4:2] bits in the Status register must be 111.

There are two software interrupts, IP1 and IP0. Software interrupts can be generated by setting the corresponding bit in the Cause register. The application program may use these bits to request interrupt service. There are corresponding bits in the Status register to mask respective software interrupts.

9.2.2 Maskable Interrupt Vectors

Maskable interrupts are vectored to the addresses shown in Table 9-5, depending on the register settings. The TX19A uses the same vector addresses for both hardware and software interrupts. When an interrupt occurs, the interrupt service routine must check the interrupt controller in order to determine the source of the interrupt, read the corresponding vector address and transfer control to it.

9.2.3 Maskable Interrupt Recognition

Maskable interrupts are taken when all of the following conditions are true:

- An interrupt request bit in the IP [7:0] field of the Cause register is set (Cause).
- The corresponding interrupt mask bit in the IM [7:0] field of the Status register is set (Status).
- The Interrupt Enable (IE) bit in the Status register is set (Status).
- The processor is not in Debug mode (Debug); i.e., the DM bit in the Debug register is cleared.
- The Error Level (ERL) and Exception Level (EXL) bits in the Status register are cleared.

For the processor to accept a maskable hardware interrupt, the IM [4:2] bits in the Status register must be 111.

In the event that both hardware- and software-requested interrupts are posted simultaneously, the hardware interrupt is delivered first while the software interrupt is left pending.

Table 9-6 Mapping of Interrupts to the Cause and Status Registers

Interrupt type	Interrupt Number	Cause Register		Status Register	
		Bit Number	Name	Bit Number	Name
Software Interrupt	0	[8]	IP0	[8]	IM0
	1	[9]	IP1	[9]	IM1
Hardware Interrupt	0	[10]	IP2	[10]	IM2
	1	[11]	IP3	[11]	IM3
	2	[12]	IP4	[12]	IM4
	3	[13]	IP5	[13]	IM5
	4	[14]	IP6	[14]	IM6
Hardware Interrupt or Timer Interrupt	5	[15]	IP7	[15]	IM7

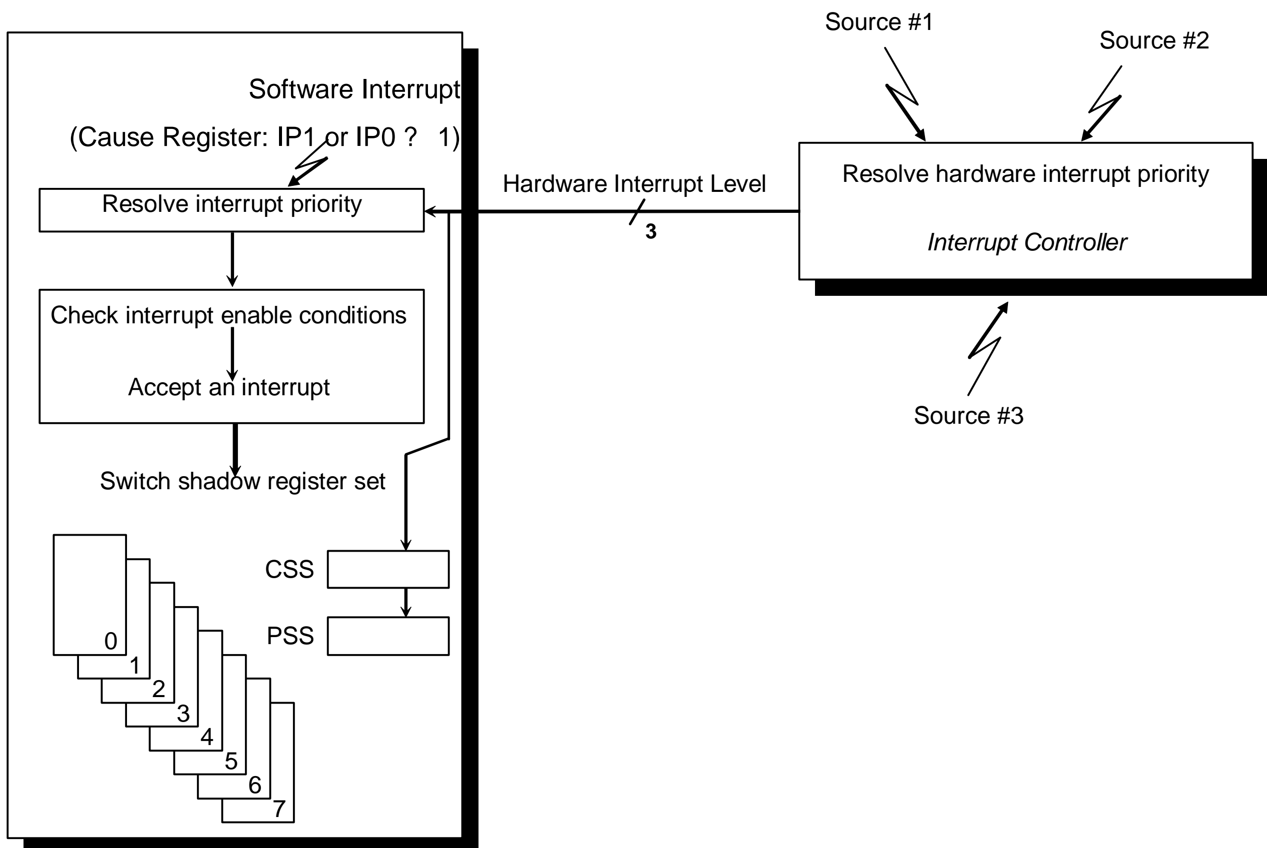


Figure 9-7 Maskable Interrupt Recognition

9.2.4 Shadow Register Sets

When a hardware interrupt occurs, the TX19A switches to a specific shadow register set associated with its priority level. The interrupt level is set to CSS bit (bit 3-0) in SSCR register. At the same time, CSS bit before update is set to PSS bit (bit11-8). The 3-bit priority-level signals from the interrupt controller are connected to the IP4, IP3 and IP2 ports of the processor. When the processor recognizes the interrupt, it switches to corresponding shadow register set depend on the signal conditions (see Table 9-8). Software interrupts, the internal timer interrupt or any other exceptions do not cause the TX19A to switch the shadow register set. The value of the PSS field is updated instead.

On execution of the ERET instruction, the value of the PSS field is restored to the CSS field (see Figure 8-1).

A debug exception and a return from a debug exception (via a DERET instruction) do not change the CSS and PSS fields.

Table 9-8 Relationships Between IP[4:2] Signals and Shadow Register Sets

IP4	IP3	IP2	Shadow Register Set
0	0	0	–
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

9.3 Debug Exceptions

There are Single-Step and Debug Breakpoint exceptions in the TX19A. This section provides details concerning sources of specific debug exceptions, how each arises and how each processed.

9.3.1 How Debug Exception Processing Work

The TX19A allows program instruction execution to arbitrarily stop to handle debugging events. Code execution breakpoints can be generated by the Software Debug Breakpoint (SDBBP) instruction. The single-step feature may be enabled by setting the SSt bit in the Debug register.

Debug exception processing occurs in the sequence shown in Figure 9-9.

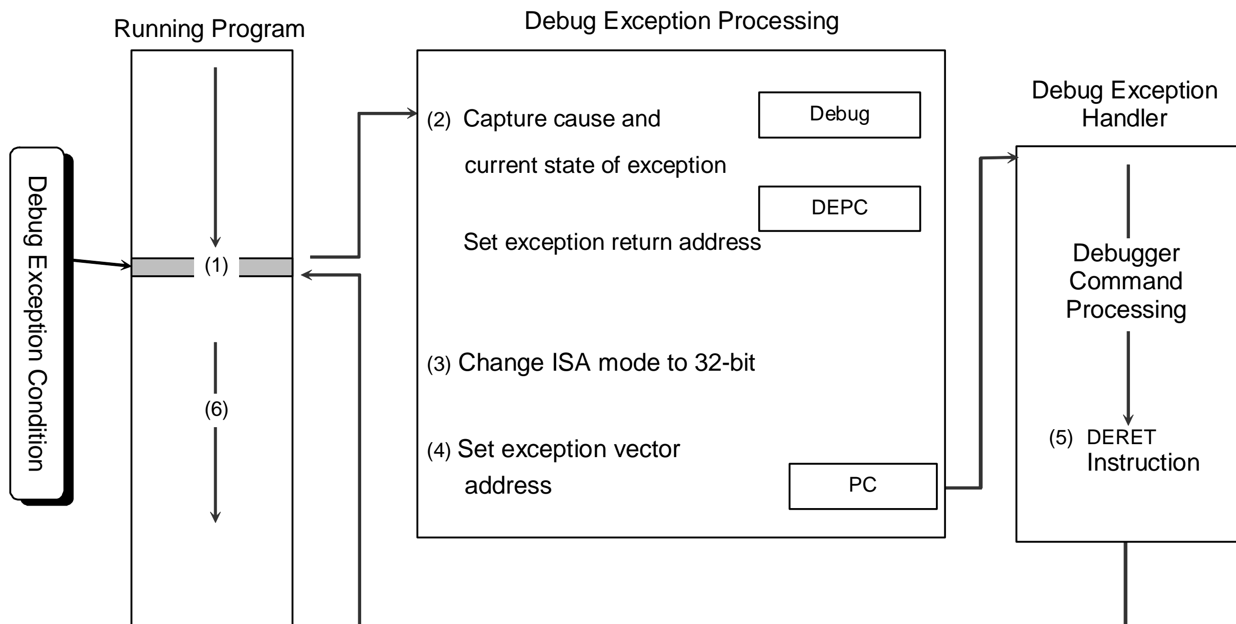


Figure 9-9 Exception Operation

5. The currently executing instruction and any subsequent instructions in the pipeline are aborted.

6. The debug exception registers save information about the debugging event.

- The Debug register shows the cause of the debug exception and whether it is currently being serviced.

- The DEPC register captures the virtual address of the instruction that caused a debug exception. When the instruction is in a jump or branch delay slot, the DEPC register is rolled back to point to the jump or branch instruction so that it can be re-executed, and the DBD bit in the Debug register is set. The least-significant bit of the DEPC register is the ISA mode bit that indicates the ISA mode that as in effect when the exception occurred.

3. The processor enters Kernel mode and disables all interrupts, independent of the setting of the Status register. If the exception occurs in 16-bit ISA mode, the least-significant bit (i.e., the ISA mode bit) of the PC is set to zero, bringing the processor into 32-bit ISA mode.
4. The PC is loaded with the Debug exception vector address to jump to the starting location of the debug exception handler.
5. At completion of the debug exception handler, the DERET instruction is executed to jump back to the return address saved in the DEPC register.
6. Processing resumes from the point where the processor left off when the exception occurred.

9.3.2 Debug Exception Types

Table 9-10 gives the types of debug exceptions that can occur in the TX19A processor.

Table 9-10 Debug Exception Types

Exception Type	Description
Single-Step	A Single-step exception occurs before the next instruction starts execution when the SSt bit in the Debug register is set.
Debug Breakpoint	A Debug Breakpoint exception provides a code execution breakpoint; it occurs when an SDBBP instruction is executed. If the SSt bit in the Debug register is set, a Single-step exception takes precedence over a Debug Breakpoint exception. If the SDBBP instruction is executed while a debug exception is being serviced (i.e., when the DM bit in the Debug register is 1), another debug exception is taken; in this case, the Break exception code is set into the DExcCode field in the Debug register.

9.3.3 Debug Exception Priorities

A debug exception and a general exception may occur simultaneously. In that case, the processor services the exceptions in the order shown in Table 9-1.

9.3.4 Exception Masking

While a debug exception is being serviced (DM=1 and IEXI=1), the processor masks all the other exceptions.

- When a Bus Error event occurs on an instruction fetch, the IBusEP bit in the Debug register is set to flag its occurrence. When a Bus Error event occurs on a data access, the DBusEP bit in the Debug register is set.
- All maskable interrupts are disabled while a debug exception is being serviced. (Maskable interrupts are unmasked by the execution of a DERET instruction.)
- A non-maskable interrupt is left pending until a return from a debug exception is made through the DERET instruction.
- When the IEXI bit in the Debug register is cleared, the TX19A responds to a general exception event (except maskable and non-maskable interrupt requests). Even if a general-exception condition arises, a debug exception is processed, causing the processor to jump to the debug exception handler. In this case, the Debug register bits that indicate the cause of the exception (DINT, DIB, DBp, DSS, DDBSImpr and DDBLImpr) remain unchanged. Instead the DExcCode field shows the cause of the general exception that occurred in Debug mode.

9.3.5 Executing a Debug Exception Handler

A debug exception handler should operate the processor under controlled conditions for program debug. It should check the DSS and DBp bits in the Debug register to determine whether to perform single-step execution or code-execution breakpoint operations.

9.3.6 Returning from Debug Exceptions

Returning from the debug exception handler is made through the DERET instruction, which performs the following:

1. Restores the return address in the DEPC register into the program counter (PC) so that the processor resumes processing from the point where a debug exception occurred. If the instruction that caused an exception is in a jump or branch delay slot, the PC points at the preceding jump or branch instruction so that it can be re-executed. The ISA mode bit of the PC is restored from bit 0 of the DEPC register to enter ISA mode that was in effect before the exception occurred.
2. Clears the Debug Mode (DM) and IEXI bits in the Debug register.
3. Gets out of the forced "Kernel mode" state.

9.3.7 Single-Step Exception

■ Cause

This exception occurs when the SSt bit in the Debug register is set.

■ Handling

A Single-step exception takes place before executing the next instruction. Figure 9-11 highlights the CP0 register fields that are used to handle this exception.

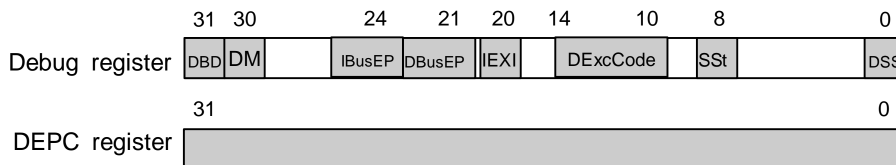


Figure 9-11 Single-Step Exception

1. The DM and DSS bits in the Debug register are set. That a Single-Step exception occurred means the SSt bit had been set.
2. The DEPC register stores the program counter on the exception. The least-significant bit in the DEPC register saves the ISA mode that was in effect prior to the exception.
3. The processor enters Kernel mode and disables all interrupts, independent of the settings of the Status register.
4. The processor jumps to the exception handler located at address 0xBFC0_0480.

The processor does not take a Single-Step exception for the following cases:

- the instruction in a jump or branch delay slot
- the first instruction on returning from a debug instruction through the DERET instruction (see Figure 9-12)
- a debug exception is being serviced (i.e., the DM bit in the Debug register is set)

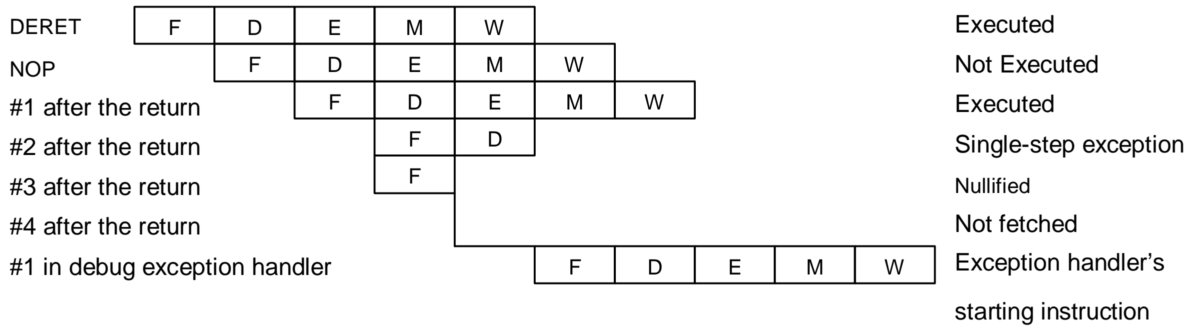


Figure 9-12 CPU Pipeline Operation After the DERET Instruction

9.3.8 Debug Breakpoint Exception

■ Cause

This exception occurs when an SDBBP instruction is executed.

■ Handling

Figure 9-13 highlights the CP0 register fields that are used to handle this exception.

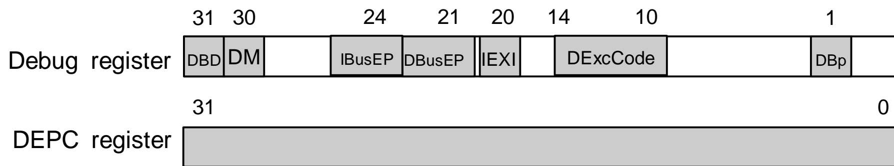


Figure 9-13 Debug Breakpoint Exception

1. The DM and DBp bits in the Debug register are set. That a Debug Breakpoint exception occurred means the SSt bit had been cleared.
2. The DEPC register stores the program counter on the exception. If the processor is executing an instruction in a jump or branch delay slot, the DEPC register points at the preceding jump or branch instruction, and the DBD bit in the Debug register is set. The least-significant bit in the DEPC register saves the ISA mode that was in effect prior to the exception.
3. The processor enters Kernel mode and disable all interrupts, independent of the settings of the Status register.
4. If the exception occurs while the processor is in 16-bit ISA mode, the processor switches to 32-bit ISA mode.
5. The processor jumps to the exception handler located at address 0xBFC0_0480.

The unused bits (bits 25-6 in the 32-bit ISA, bits 10-5 in the 16-bit ISA) in an SDDBP instruction are available for use as software parameters to pass additional information to an exception handler. To examine these bits, load the contents of the instruction at which the DEPC register points. If the instruction is in a jump or branch delay slot (i.e., the DBD bit in the Debug register is set), add four to the contents of the DEPC register to locate the instruction.

To resume execution after the exception has been serviced, alter the contents of the DEPC register by adding four (in 32-bit ISA mode) or two (in 16-bit ISA mode) so that the SDDBP instruction is not re-executed. If the SDDBP instruction is in a jump or branch delay slot (i.e., the DBD bit in the Debug register is set), the instruction at the return address is a jump or branch instruction. In that case, the jump or branch instruction must be interpreted to set the DEPC register before resuming execution.

Chapter 10 Power Consumption Management

The TX19A provides hardware support for several levels of power reduction. The Halt and Doze modes are entered by setting the RP bit in the CP0's Status register and executing the WAIT instruction. This chapter describes the power management features and capabilities provided by the TX19A.

10.1 Power-Saving Modes

Figure 10-1 illustrates the power-saving modes provided by the TX19A.

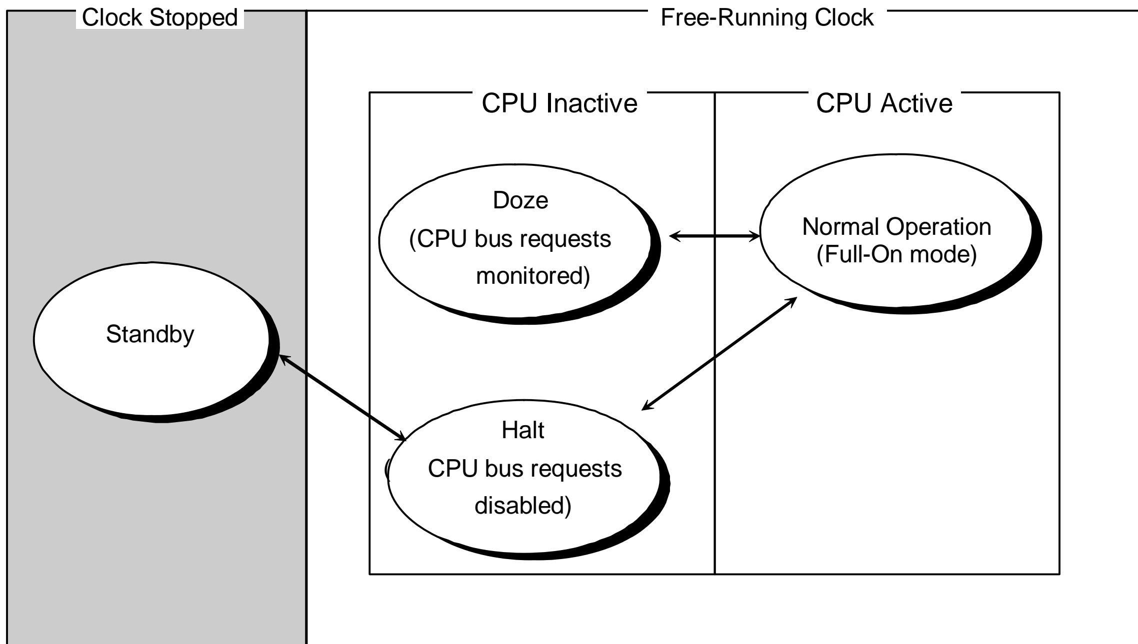


Figure 10-1 Power-Saving Modes

The TX19A has the capability to dynamically control power consumption during operation. Table 10-2 describes the available power-saving modes.

Table 10-2 Power-Saving Modes

Mode	Description
Standby Mode	For lowest power operation, the processor clock can be removed altogether. There are two levels of power savings achieved through Standby mode. 1. In one mode, both the processor and the oscillator circuitry are disabled altogether. 2. In the other mode, the oscillator circuitry continues to run, but the clock input to the processor is disabled. For details on Standby mode, see respective hardware user's manuals.
Halt Mode	In Halt mode, all activities of the processor stop, and the CPU bus monitoring is disabled. The TX19A processor assumes bus mastership. Halt mode can be entered by executing the WAIT instruction when the RP bit in the Status register is cleared.
Doze Mode	In Doze mode, all activities of the processor stop except for the CPU bus monitor that continues to operate and recognizes bus requests. Bus mastership is granted to an external agent. Doze mode can be entered by executing the WAIT instruction when the RP bit in the Status register is set.
Normal Mode (Full-On Mode)	This is the default power state of the TX19A following a hardware reset, with the processor fully powered and operating at full clock speed.
Other Modes	There are components having additional power-saving capabilities, e.g., a very-low speed mode in which the clock runs at 32.768 kHz for time-of-day clocks. For additional power-saving modes, see respective hardware user's manuals.

10.2 Halt Mode

Figure 10-2 depicts how Halt mode can be entered.

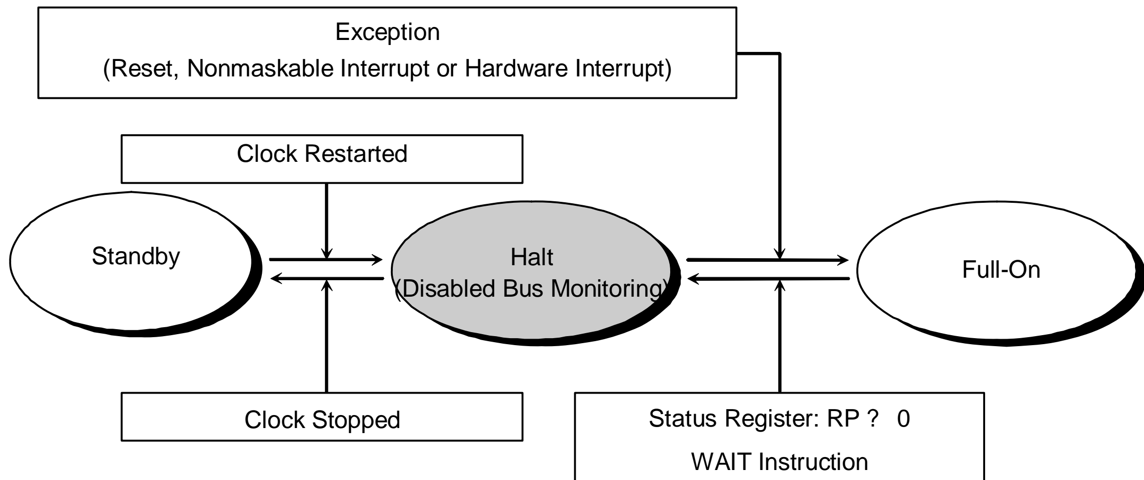


Figure 10-2 Halt Mode

The processor enters Halt mode on execution of the WAIT instruction when the RP bit in the Status register is cleared during normal operation mode. Halt mode freezes the "processor core," preserving the pipeline state. In Halt mode, the processor ignores any external bus requests, as it monopolizes mastership of the bus.
In Halt mode, the on-chip write buffer unit (if any) continues to operate until all entries in it have been written to external memory.

A wakeup from Halt mode can be achieved by causing a Reset, Nonmaskable
Interrupt or Maskable Hardware Interrupt exception. Any of these exceptions causes the processor to exit Halt mode and take an exception.

Maskable interrupts are recognized even if they are masked in the Status register. In that case, after a wakeup, normal processing resumes with all register contents intact, i.e., the processor continues execution from the address following the instruction that brought the processor into Halt mode.

In Halt mode, the processor may have its clock input shut down for additional power savings. The oscillator and/or clock stop causes the processor to enter Standby mode. Restarting the clock to the processor causes it to return to Halt mode.

10.3 Doze Mode

Figure 10-3 depicts how Doze mode can be entered.

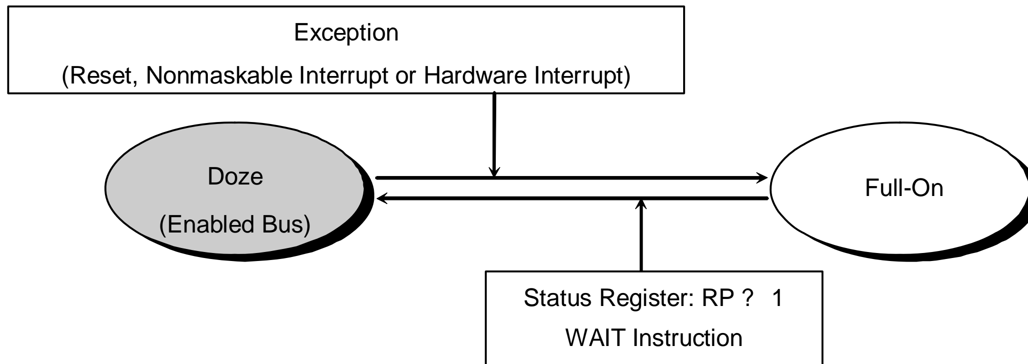


Figure 10-3 Doze Mode

The processor enters Doze mode on execution of the WAIT instruction when the RP bit in the Status register is cleared during normal operation mode. Like Halt mode, Doze mode freezes the "processor core," preserving the pipeline state, but in Doze mode, the processor recognizes external bus requests. In Doze mode, the on-chip write buffer unit (if any) continues to operate until all entries in it have been written to external memory.

A wakeup from Doze mode can be achieved by causing a Reset, Non-maskable Interrupt or Maskable Hardware Interrupt exception. Any of these exceptions causes the processor to exit Doze mode and take an exception.

Maskable interrupts are recognized even if they are masked in the Status register. In that case, after a wakeup, normal processing resumes with all register contents intact, i.e., the processor continues execution from the address following the instruction that brought the processor into Doze mode.

Appendix A 32-Bit ISA Details

This appendix presents detailed information concerning each instruction in the 32-bit ISA, including assembler syntax, instruction format, operation and exceptions that may occur due to the execution of the instruction. Each instruction is listed alphabetically by mnemonic. For the variations of instruction formats, see Section 3.1, *Instruction Formats*.

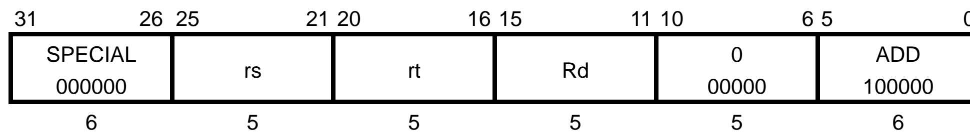
ADD *rd, rs, rt*

Add

Operation

$rd \Leftarrow rs + rt$

Instruction Encoding



Description

The contents of general-purpose register *rs* is added to the contents of general-purpose register *rt*, and the result is placed into general-purpose register *rd*.

In the case of c←a+b, an Interger Overflow exception occurs if a and b has the same sign and c has the different one. The destination register (*rd*) is not altered when an Integer Overflow exception occurs.

Exceptions

Interger Overflow exception

Examples

1. Assume that registers r2 and r3 contain 0x0200_0000 and 0x0123_4567 respectively. Then, executing the instruction:

```
ADD r4,r2,r3
```

places the sum (0x0323_4567) into r4.

2. Assume that registers r2 and r3 contain 0x7FFF_FFFF and 0x0000_0001 respectively. Then, the addition of r2 and r3 gives the result 0x8000_0000, which is a negative number, indicating a 2's-complement overflow. Thus executing the instruction:

```
ADD r4,r2,r3
```

causes an Integer Overflow exception. Register r4 is not modified as a result of this instruction.

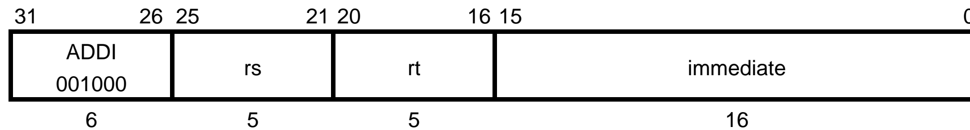
ADDI *rt, rs, immediate*

Add Immediate

Operation

$rt \Leftarrow rs + ((immediate_{15})^{16} \parallel immediate_{15..0})$

Instruction Encoding



Description

The 16-bit *immediate* is sign-extended and added to the contents of general-purpose register *rs*. The result is placed into general-purpose register *rt*.

An Integer Overflow exception is taken on 2's-complement overflow. The destination register (*rt*) is not altered when an Integer Overflow exception occurs.

With the 16-bit signed *immediate*, the immediate range is -32768 to +32767. If a number is outside this range, you need to put it in a general-purpose register and use the ADD or ADDU instruction (see Section 3.3.2, *32-Bit Constants*).

Exceptions

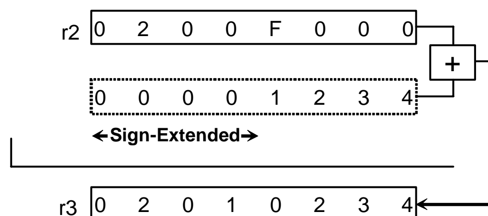
Integer Overflow exception

Example

Assume that register r2 contains 0x0200_F000. Then, executing the instruction:

```
ADDI r3,r2,0x1234
```

places the sum 0x0201_0234 into r3.



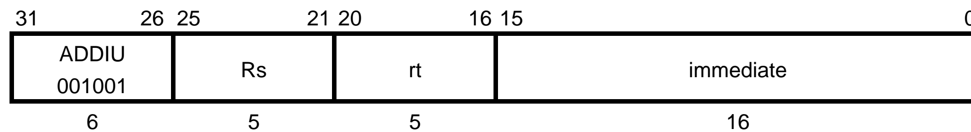
ADDIU *rt, rs, immediate*

Add Immediate Unsigned

Operation

$rt \Leftarrow rs + ((immediate_{15})^{16} \,||\, immediate_{15..0})$

Instruction Encoding



Description

The term "Add Immediate Unsigned" is a misnomer; the 16-bit *immediate* is *sign-extended* and added to the contents of general-purpose register *rs*. The result is placed into general-purpose register *rt*. The only difference between this instruction and the ADDI instruction is that this instruction never causes an Integer Overflow exception.

Exceptions

None

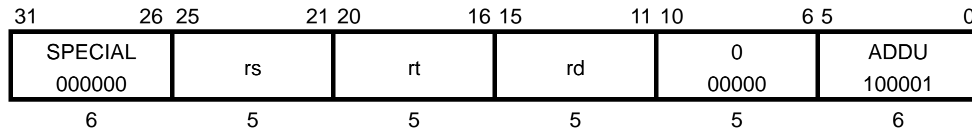
ADDU *rd*, *rs*, *rt*

Add Unsigned

Operation

$rd \Leftarrow rs + rt$

Instruction Encoding



Description

The contents of general-purpose register *rs* is added to the contents of general-purpose register *rt*, and the result is placed into general-purpose register *rd*.
The only difference between this instruction and the ADD instruction is that this instruction never causes an Integer Overflow exception.

Exceptions

None

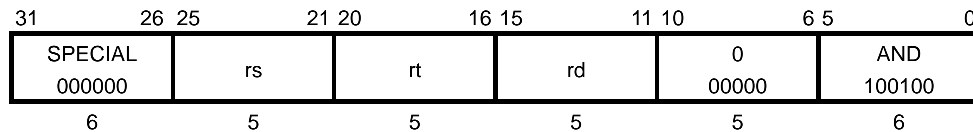
AND *rd*, *rs*, *rt*

AND

Operation

$rd \Leftarrow rs$ AND rt

Instruction Encoding



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

Description

The contents of general-purpose register *rs* is ANDed with the contents of general-purpose register *rt*, and the result is placed into general-purpose register *rd*.

Exceptions

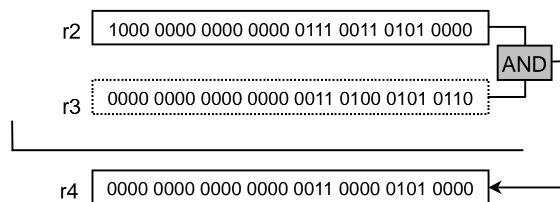
None

Example

Assume that registers r2 and r3 contain 0x8000_7350 and 0x0000_3456 respectively. Then, the instruction:

```
AND r4,r2,r3
```

performs the logical AND between r2 and r3 and puts the result (0x0000_3050) in r4, as shown below.



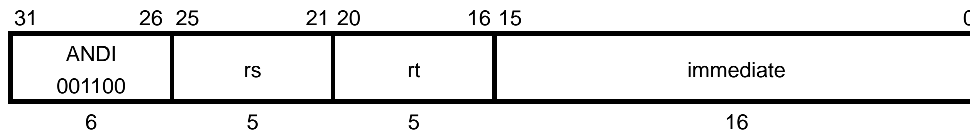
ANDI *rt*, *rs*, *immediate*

Logical AND Immediate

Operation

$rt \Leftarrow rs$ AND $(0^{16} \parallel immediate_{15..0})$

Instruction Encoding



Description

The 16-bit *immediate* is zero-extended and ANDed with the contents of general-purpose register *rs*. The result is placed into general-purpose register *rt*.

The *immediate* field is 16 bits in length. If the *immediate* size is larger than that, you need to put it in a general-purpose register and use the AND instruction (see Section 3.3.2, *32-Bit Constants*).

Exceptions

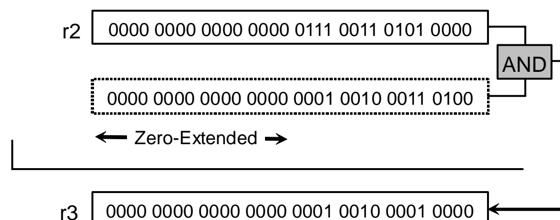
None

Example

Assume that register r2 contains 0x0000_7350. Then, the instruction:

```
ANDI r3,r2,0x1234
```

performs the logical AND between 0x0000_7350 and 0x0000_1234 and puts the result (0x0000_1210) in r3, as shown below.



B *offset*

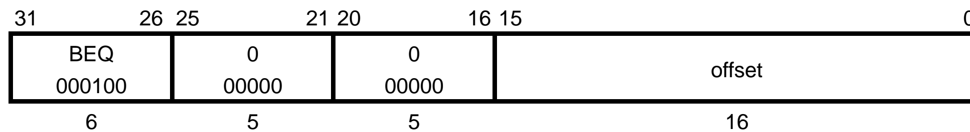
Assembly Idiom

Unconditional Branch

Operation

$pc \Leftarrow pc + 4 + \text{sign-extend}(\textit{offset} \parallel 00)$

Instruction Encoding



Description

The program unconditionally branches to the target address with a delay of one instruction (or two pipeline cycles). The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

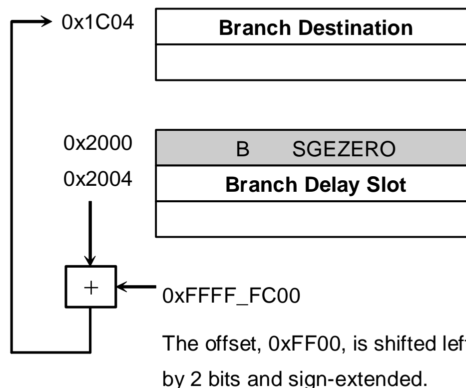
None

Example

```
B SGEZERO
```

Assume that the B instruction resides at address 0x2000 and that label SGEZERO points to absolute address 0x1C04. Then the assembler/linker turns this label into a relative offset of 0xFF00 (see the figure below).

The processor unconditionally transfers program control to 0x1C04. The instruction in the branch delay slot is executed before the branch is taken.



BAL *offset*

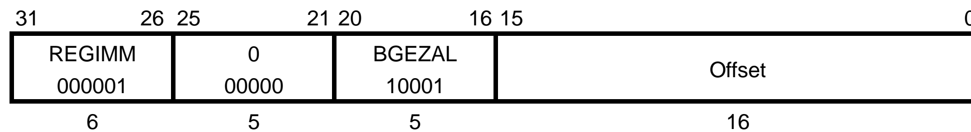
Assembly Idiom

Branch And Link

Operation

r31 ⇐ pc + 8; pc ⇐ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

The program unconditionally branches to the target address with a delay of one instruction (or two pipeline cycles). The address of the instruction after the branch delay slot (PC+8) is saved in the link register, r31 (ra). The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

None

Example

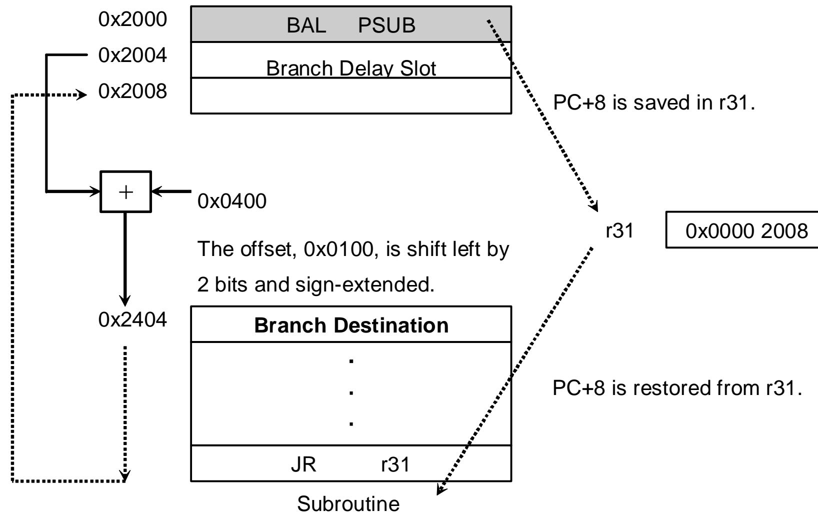
```
BAL PSUB
```

Assume that the BAL instruction resides at address 0x2000 and that label PSUB points to absolute address 0x2404. Then the assembler/linker turns this label into a relative offset of 0x0100 (see the figure below).

The processor unconditionally transfers program control to address 0x2404. The instruction in the branch delay slot is executed before the branch is taken.

The JR instruction is used at the end of the called subroutine to return control to the instruction after the branch delay slot (PC+8).

```
JR r31
```


0x2000
BAL PSUB
0x2004
Branch Delay Slot
0x2008
PC+8 is saved in r31.
+
0x0400
r31
0x0000 2008
The offset, 0x0100, is shift left by
2 bits and sign-extended.
0x2404
Branch Destination
PC+8 is restored from r31.
JR r31
Subroutine

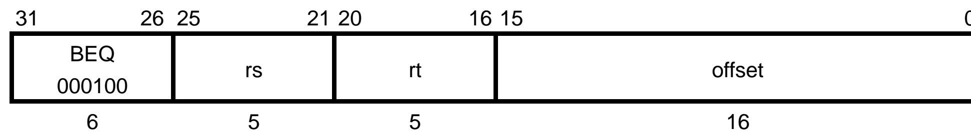
BEQ *rs*, *rt*, *offset*

Branch On Equal

Operation

if *rs* = *rt* then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

The contents of general-purpose register *rs* is compared to the contents of general-purpose register *rt*. If the two registers are equal, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). The instruction in the branch delay slot is always executed, regardless of whether the branch is taken or not. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

None

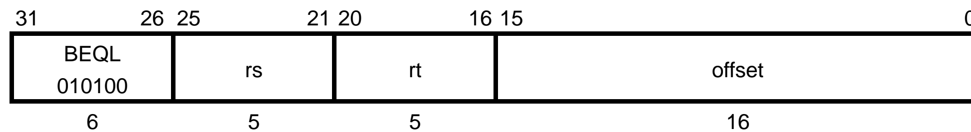
BEQL *rs*, *rt*, *offset*

Branch On Equal Likely

Operation

if *rs* = *rt* then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

The contents of general-purpose register *rs* is compared to the contents of general-purpose register *rt*. If the two registers are equal, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). If the branch condition is true, the instruction in the branch delay slot is executed before the branch; otherwise, it is nullified. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

None

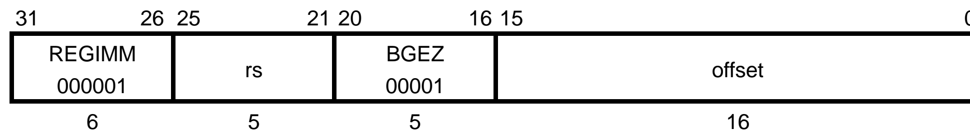
BGEZ *rs*, *offset*

Branch On Greater Than Or Equal To Zero

Operation

if $rs \geq 0$ then pc $\Leftarrow$ pc + 4 + sign-extend(*offset* $\|$ 00)

Instruction Encoding



Description

If the contents of general-purpose register *rs* are greater than or equal to zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). The instruction in the branch delay slot is always executed, regardless of whether the branch is taken or not. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

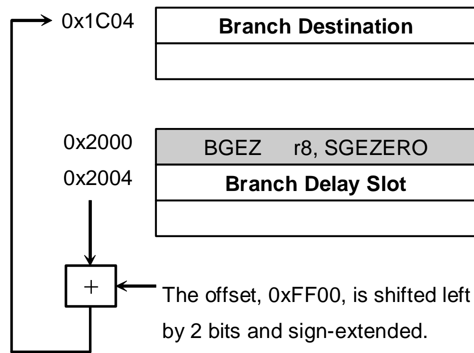
None

Example

```
BGEZ r8,SGEZERO
```

Assume that this branch instruction resides at address 0x2000 and that label SGEZERO points to absolute address 0x1C04. Then the assembler/linker turns this label into a relative offset of 0xFF00 (see the figure below).

If the contents of r8 is greater than or equal to zero (i.e., r8 has the sign bit cleared), the processor transfers program control to address 0x1C04. The branch takes effect after the instruction in the branch delay slot is executed.


0x1C04
Branch Destination
0x2000
BGEZ r8, SGEZERO
0x2004
Branch Delay Slot
+
The offset, 0xFF00, is shifted left
by 2 bits and sign-extended.

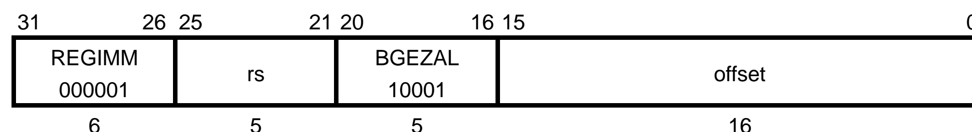
BGEZAL *rs*, *offset*

Branch On Greater Than or Equal To Zero And Link

Operation

r31 $\Leftarrow$ pc +8; if *rs* $\geq$ 0 then pc $\Leftarrow$ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

If the contents of general-purpose register *rs* is greater than or equal to zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles), and saves the address of the instruction following the branch delay slot (PC+8) in the link register, r31. The instruction in the branch delay slot is always executed, regardless of whether the branch is taken or not. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

General-purpose register *rs* may not be r31 because such an instruction cannot be restarted, with the contents of *rs* altered by the return address. An exception or interrupt could prevent the completion of a legal instruction in the branch delay slot. If that happens, after the exception handler routine has been executed, processing must restart with the branch instruction.

Exceptions

None

Example

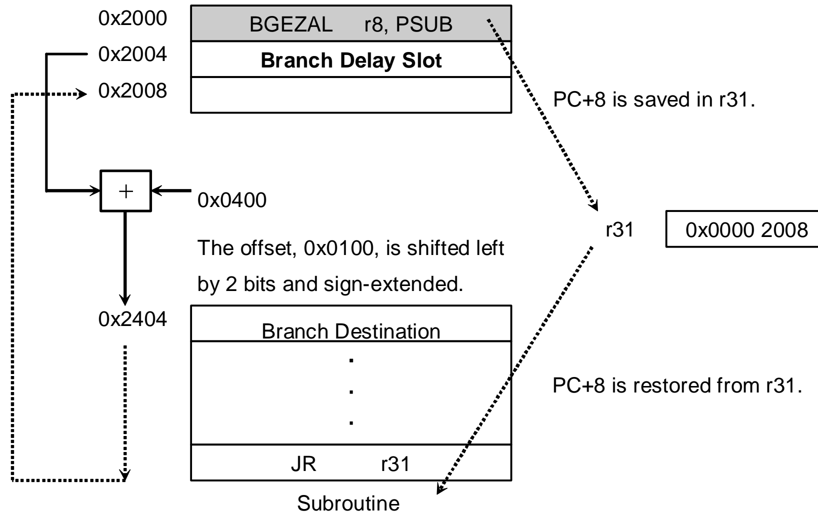
```
BGEZAL r8,PSUB
```

Assume that this branch instruction resides at address 0x2000 and that label PSUB points to absolute address 0x2404. Then the assembler/linker turns this label into a relative offset of 0x0100 (see the figure below).

If the contents of r8 is greater than or equal to zero (i.e., r8 has the sign bit cleared), the processor transfers program control to address 0x2404. The branch takes effect after the instruction in the branch delay slot is executed.

The JR instruction is used at the end of the called subroutine to return control to the instruction after the branch delay slot (PC+8).

```
JR r31
```



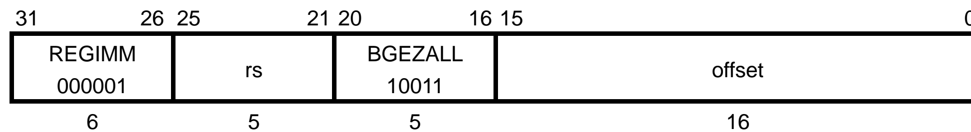
BGEZALL *rs*, *offset*

Branch On Greater Than Or Equal To Zero And Link Likely

Operation

r31 ⇐ pc +8; if *rs* ≥ 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

If the contents of general-purpose register *rs* is greater than or equal to zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles), and saves the address of the instruction following the branch delay slot (PC+8) in the link register, r31. If the branch condition is true, the instruction in the branch delay slot is executed before the branch; otherwise, it is nullified. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

General-purpose register *rs* may not be r31 because such an instruction cannot be restarted, with the contents of *rs* altered by the return address. An exception or interrupt could prevent the completion of a legal instruction in the branch delay slot. If that happens, after the exception handler routine has been executed, processing must restart with the branch instruction.

Exceptions

None

Example

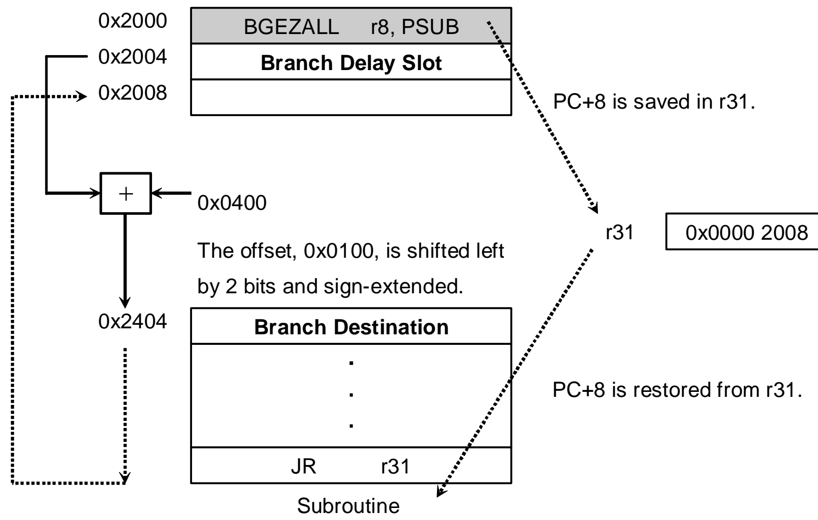
```
BGEZALL r8,PSUB
```

Assume that this branch instruction resides at address 0x2000 and that label PSUB points to absolute address 0x2404. Then the assembler/linker turns this label into a relative offset of 0x0100.

If the contents of r8 is greater than or equal to zero (i.e., r8 has the sign bit cleared), the processor transfers program control to address 0x2404. The branch takes effect after the instruction in the branch delay slot is executed. When the branch is not taken, the instruction in the branch delay is nullified.

The JR instruction is used at the end of the called subroutine to return control to the instruction after the branch delay slot (i.e., PC+8).

```
JR r31
```



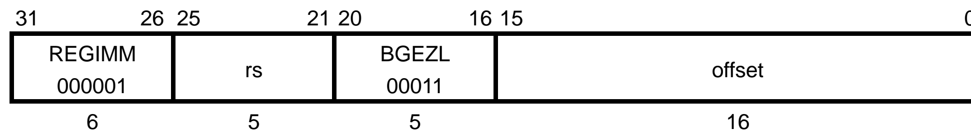
BGEZL *rs*, *offset*

Branch On Greater Than Or Equal To Zero Likely

Operation

if $rs \geq 0$ then pc $\Leftarrow$ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

If the contents of general-purpose register *rs* is greater than or equal to zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). If the branch condition is true, the instruction in the branch delay slot is executed before the branch; otherwise, it is nullified. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

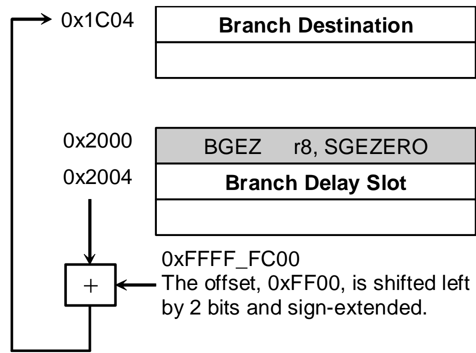
None

Example

```
BGEZL r8,SGEZERO
```

Assume that this branch instruction resides at address 0x2000 and that label SGEZERO points to absolute address 0x1C04. Then the assembler/linker turns this label into a relative offset of 0xFF00 (see the figure below).

If the contents of r8 is greater than or equal to zero (i.e., r8 has the sign bit cleared), the processor transfers program control to address 0x1C04. The branch takes effect after the instruction in the branch delay slot is executed. When the branch is not taken, the instruction in the branch delay slot is nullified.


0x1C04
Branch Destination
0x2000
BGEZ r8, SGEZERO
0x2004
Branch Delay Slot
+
0xFFFF_FC00
The offset, 0xFF00, is shifted left
by 2 bits and sign-extended.

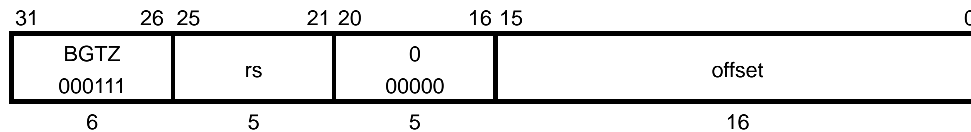
BGTZ *rs*, *offset*

Branch On Greater Than Zero

Operation

if *rs* > 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

If the contents of general-purpose register *rs* is greater than zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). The instruction in the branch delay slot is always executed, regardless of whether the branch is taken or not. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

None

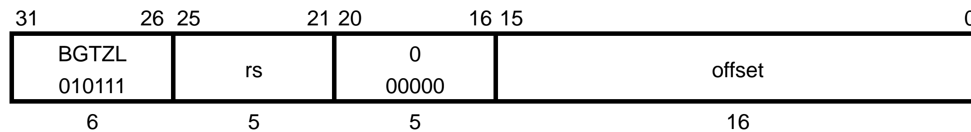
BGTZL *rs*, *offset*

Branch On Greater Than Zero Likely

Operation

if *rs* > 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

If the contents of general-purpose register *rs* is greater than zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). If the branch condition is true, the instruction in the branch delay slot is executed before the branch; otherwise, it is nullified. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

None

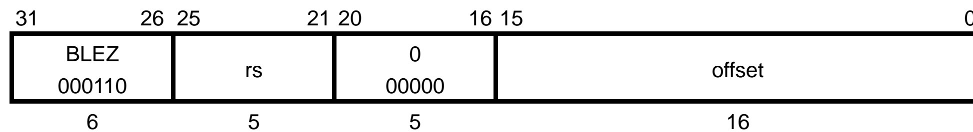
BLEZ *rs*, *offset*

Branch On Less Than Or Equal To Zero

Operation

if *rs* ≤ 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

If the contents of general-purpose register *rs* is less than or equal to zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). The instruction in the branch delay slot is always executed, regardless of whether the branch is taken or not. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

None

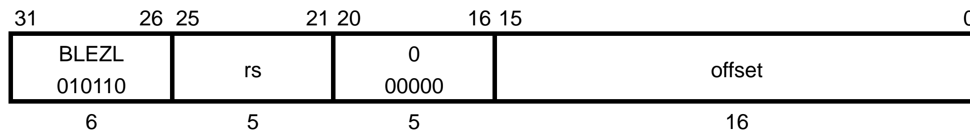
BLEZL *rs*, *offset*

Branch On Less Than Or Equal To Zero Likely

Operation

if *rs* ≤ 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

If the contents of general-purpose register *rs* is less than or equal to zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). If the branch condition is true, the instruction in the branch delay slot is executed before the branch; otherwise, it is nullified. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

None

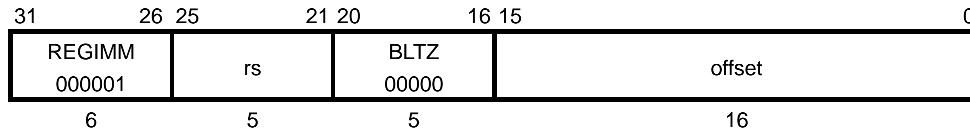
BLTZ *rs*, *offset*

Branch On Less Than Zero

Operation

if *rs* < 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

If the contents of general-purpose register *rs* is less than zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). The instruction in the branch delay slot is always executed, regardless of whether the branch is taken or not. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

None

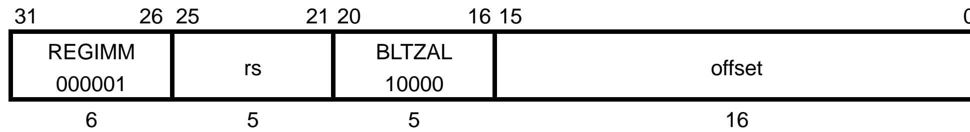
BLTZAL *rs*, *offset*

Branch On Less Than Zero And Link

Operation

r31 ⇐ pc +8; if *rs* < 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

If the contents of general-purpose register *rs* is less than zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). The instruction in the branch delay slot is always executed, regardless of whether the branch is taken or not. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address. The address of the instruction following the branch delay slot (PC+8) is unconditionally saved in the link register, r31.

General-purpose register *rs* may not be r31 because such an instruction is not restart able, with the contents of *rs* altered by the return address. An exception or interrupt could prevent the completion of a legal instruction in the branch delay slot. If that happens, after the exception handler routine has been executed, processing must restart with the branch instruction.

Exceptions

None

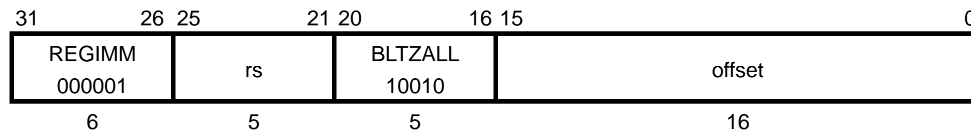
BLTZALL *rs*, *offset*

Branch On Less Than Zero And Link Likely

Operation

r31 ⇐ pc +8; if *rs* < 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

If the contents of general-purpose register *rs* is less than zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles), and saves the address of the instruction following the branch delay slot (PC+8) in the link register, r31. If the branch condition is true, the instruction in the branch delay slot is executed before the branch; otherwise, it is nullified. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

General-purpose register *rs* may not be r31 because such an instruction cannot be restarted, with the contents of *rs* altered by the return address. An exception or interrupt could prevent the completion of a legal instruction in the branch delay slot. If that happens, after the exception handler routine has been executed, processing must restart with the branch instruction.

Exceptions

None

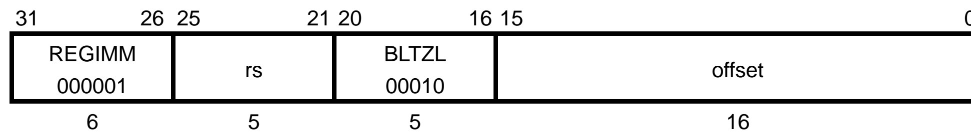
BLTZL *rs*, *offset*

Branch On Less Than Zero Likely

Operation

if *rs* < 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

Instruction Encoding



Description

If the contents of general-purpose register *rs* is less than zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). If the branch condition is true, the instruction in the branch delay slot is executed before the branch; otherwise, it is nullified. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

None

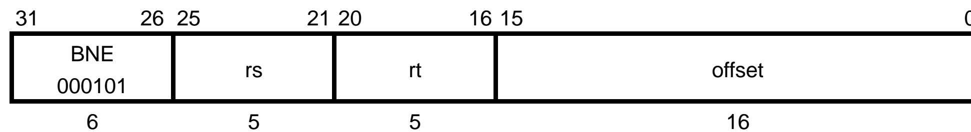
BNE *rs*, *rt*, *offset*

Branch On Not Equal

Operation

if $rs \neq rt$ then pc $\Leftarrow$ pc + 4 + sign-extend(*offset* $\|$ 00)

Instruction Encoding



Description

The contents of general-purpose register *rs* is compared to the contents of general-purpose register *rt*. If the two registers are not equal, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). The instruction in the branch delay slot is always executed, regardless of whether the branch is taken or not. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

None

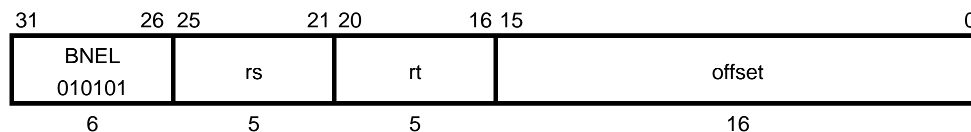
BNEL *rs*, *rt*, *offset*

Branch On Not Equal Likely

Operation

if $rs \neq rt$ then pc $\Leftarrow$ pc + 4 + sign-extend($offset$ || 00)

Instruction Encoding



Description

The contents of general-purpose register *rs* is compared to the contents of general-purpose register *rt*. If the two registers are not equal, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). If the branch condition is true, the instruction in the branch delay slot is executed before the branch; otherwise, it is nullified. The target address is computed relative to the address of the instruction in the branch delay slot (PC+4); the 16-bit immediate *offset* is shifted left by two bits, sign-extended and added to PC+4 to form the target address.

Exceptions

None

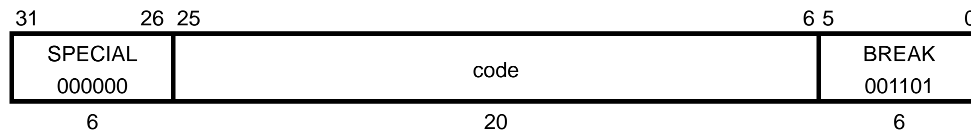
BREAK *code*

Breakpoint

Operation

Breakpoint exception

Instruction Encoding



Description

When this instruction is executed, a Breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.

The *code* field in the BREAK instruction is available for use as software parameters to pass additional information. The exception handler can retrieve it by loading the contents of the memory word containing the instruction. For more on this, see Section 9.1.11, *Breakpoint Exception*.

Exceptions

Breakpoint exception

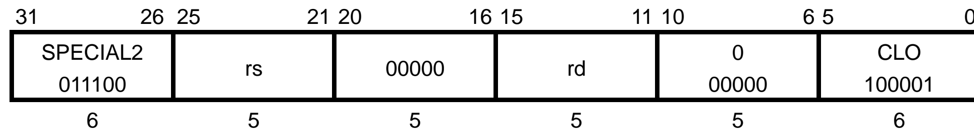
CLO *rd*, *rs*

Count Leading Ones in Word

Operation

$rd \Leftarrow$ count_leading_ones rs

Instruction Encoding



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

Description

The contents of general-purpose register *rs* is scanned from bit 31 to bit 0, and the number of leading ones is written to general-purpose register *rd*. If all 32 bits in *rs* are set, the result written to *rd* is 32.

Exceptions

None

Example

Assume that register r2 contains 0xFE23_DE67. Then, the instruction:

```
CLO     r4, r2
```

counts the number of leading ones in r2 and puts the result, 0x0000_0007, in r4.

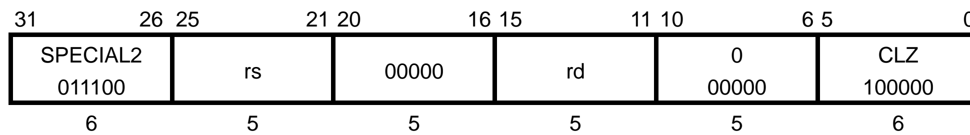
CLZ *rd*, *rs*

Count Leading Zeros in Word

Operation

rd ⇐ count_leading_zeros *rs*

Instruction Encoding



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

Description

The contents of general-purpose register *rs* is scanned from bit 31 to bit 0, and the number of leading zeros is written to general-purpose register *rd*. If all 32 bits in *rs* are set, the result written to *rd* is 32.

Exceptions

None

Example

Assume that register r2 contains 0x07EF_45CD. Then, the instruction:

```
CLZ     r4, r2
```

counts the number of leading zeros in r2 and puts the result, 0x0000_0005, in r4.

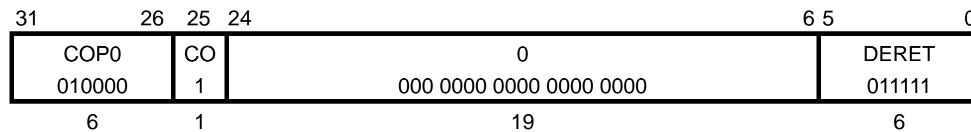
DERET

Debug Exception Return

Operation

pc ⇐ DEPC, Debug[DM] ⇐ 0, Debug[IEXI] ⇐ 0

Instruction Encoding



Description

The DERET instruction is used to return control from a debug exception handler to a user program. This is accomplished by loading the contents of the DEPC register into the program counter (PC). See Section 9.3.6, *Returning from Debug Exceptions*, for details.

The DERET instruction does not have a delay slot. It is executed with a delay of one instruction (or two pipeline cycles).

The DERET instruction restores the ISA mode bit (bit 0) of the PC from bit 0 of the DEPC register, bringing the processor into the ISA mode that had been in effect before the debug exception was taken.

The DERET instruction may not be in a jump or branch delay slot.

The operation of the DERET instruction is unpredictable if the processor is not in Debug mode (i.e., if the DM bit in the Debug register is cleared).

Typically, the DEPC register automatically captures the address of the exception-causing instruction on a debug exception. If you want to use the MTC0 instruction to load the DEPC register with a return address, the debug exception handler must execute at least two instructions before issuing the DERET instruction.

Exceptions

None

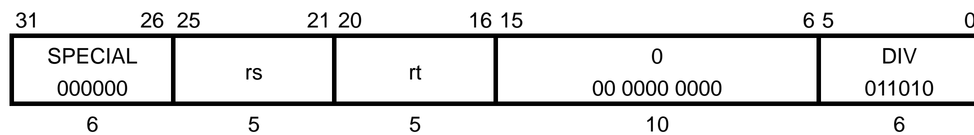
DIV *rs*, *rt*

Divide

Operation

LO ⇐ *rs* ÷ *rt*;
HI ⇐ *rs* MOD *rt*

Instruction Encoding



31–26	25–21	20–16	15–6	5–0
SPECIAL 000000	rs	rt	0 00 0000 0000	DIV 011010
6	5	5	10	6

Description

The contents of general-purpose register *rs* is divided by the contents of general-purpose register *rt*. Both operands are treated as signed integers. The quotient is placed into register LO and the remainder is placed into register HI. The DIV instruction never causes an Integer Overflow exception.

The result of the DIV instruction is undefined if the divisor is zero. Typically, it is necessary to check for a zero divisor and an overflow condition after a DIV instruction.

Any divide instruction is transferred to the dedicated divide unit as remaining instructions continue through the pipeline. The divide unit keeps running even when delay cycles and exceptions occur.

If the DIV instruction is followed by an MFHI, MFLO, MADD, MADDU, MSUB or MSUBU instruction before the quotient and the remainder are available, the pipeline stalls until they do become available (see Section 5.4, *Divide Instructions*).

Exceptions

None

DIVU *rs*, *rt*

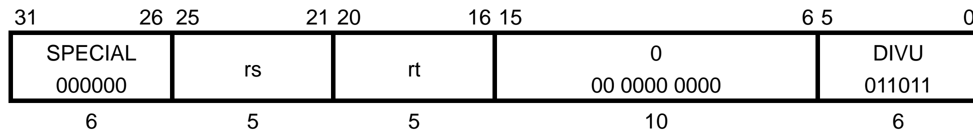
Divide Unsigned

Operation

LO ⇐ *rs* ÷ *rt*;

HI ⇐ *rs* MOD *rt*

Instruction Encoding



Description

The contents of general-purpose register *rs* is divided by the contents of general-purpose register *rt*. The quotient is placed into register LO and the remainder is placed into register HI. The DIVU instruction never causes an Integer Overflow exception. The only difference between the DIV instruction and this instruction is that this instruction treats both operands as unsigned integers.

Exceptions

None

ERET

Exception Return

Operation

if Status[ERL] = 1 then pc ⇐ ErrorEPC

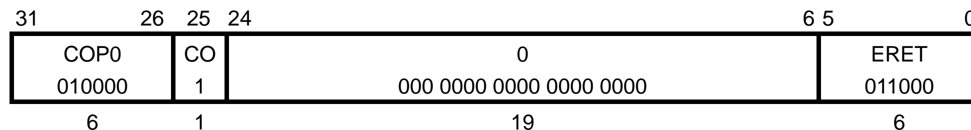
Status[ERL] ⇐ 0

else pc ⇐ EPC

Status[EXL] ⇐ 0

SSCR[CSS] ⇐ SSCR[PSS]

Instruction Encoding



Description

ERET is an instruction for returning from an interrupt, exception or error trap.

The ERET instruction does not have a delay slot. It is executed with a delay of one instruction (two pipeline cycles).

The ERET instruction restores the ISA mode bit (bit 0) of the PC from bit 0 of the ErrorEPC register, bringing the processor into the ISA mode that had been in effect before the exception was taken.

An attempt to execute the ERET instruction in User mode when the CU0 bit in the Status register is cleared causes a Coprocessor Unusable exception. If you want to use the MTC0 instruction to load the ErrorEPC or EPC register with a return address or if you have modified the contents of the Status register, the exception handler must execute at least two instructions before issuing the ERET instruction.

If the ERL bit in the Status register is set, ERET restores the PC from the ErrorPC register and then clears the ERL bit. Otherwise, ERET restores the PC from the EPC register and then clears the EXL bit.

Also, the PSS field in the SSCR register is popped to the CSS field.
ERET must not be placed in a branch or jump delay slot.

Exceptions

Coprocessor Unusable exception

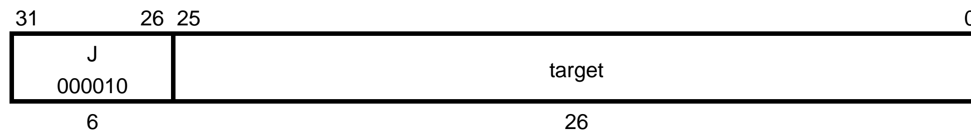
J *target*

Jump

Operation

pc ⇐ pc[31:28] || *target* || 00

Instruction Encoding



Description

The program unconditionally jumps to the target address with a delay of one instruction (or two pipeline cycles). The target address is computed relative to the address of the instruction in the jump delay slot (PC+4). The 26-bit *target* is shifted left by two bits and combined with the four most-significant bits of PC+4 to form the target address.

With the J instruction, the address of the target must be within a 2^{28}-byte segment. To jump to an arbitrary 32-bit address, load the desired address into a register and use the JR instruction (see Section 3.4.6, *Jumping to 32-Bit Addresses*).

Exceptions

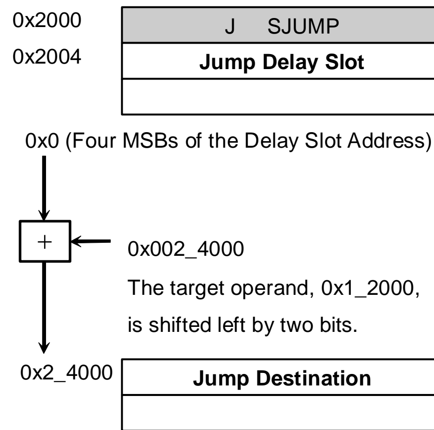
None

Example

```
J  SJUMP
```

Assume that this jump instruction resides at address 0x2000 and that label SJUMP points to absolute address 0x2_4000. Then the assembler/linker turns this label into target operand 0x1_2000 (see the figure below).

The processor unconditionally transfers program control to address 0x2_4000. The jump takes effect after the instruction in the jump delay slot is executed.


0x2000
J SJUMP
0x2004
Jump Delay Slot
0x0 (Four MSBs of the Delay Slot Address)
+
0x002_4000
The target operand, 0x1_2000,
is shifted left by two bits.
0x2_4000
Jump Destination

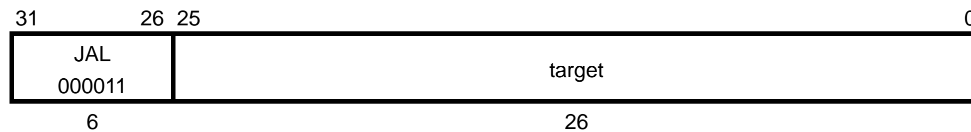
JAL *target*

Jump And Link

Operation

r31 ⇐ pc + 8; pc ⇐ pc[31:28] || *target* || 00

Instruction Encoding



Description

The program unconditionally jumps to the target address with a delay of one instruction (or two pipeline cycles). The target address is computed relative to the address of the instruction in the jump delay slot (PC+4). The 26-bit *target* is shifted left by two bits and combined with the four most-significant bits of PC+4 to form the target address. The JAL instruction never toggles the ISA mode bit of the program counter (PC).

The address of the instruction after the jump delay slot (PC+8) is saved in the link register, r31 (ra). The least-significant bit of r31 stores the ISA mode bit that was in effect before the jump.

With the JAL instruction, the address of the target must be within a 2^{28}-byte segment. To jump to an arbitrary 32-bit address, load the desired address into a register and use the JALR instruction (see Section 3.4.6, *Jumping to 32-Bit Addresses*).

Exceptions

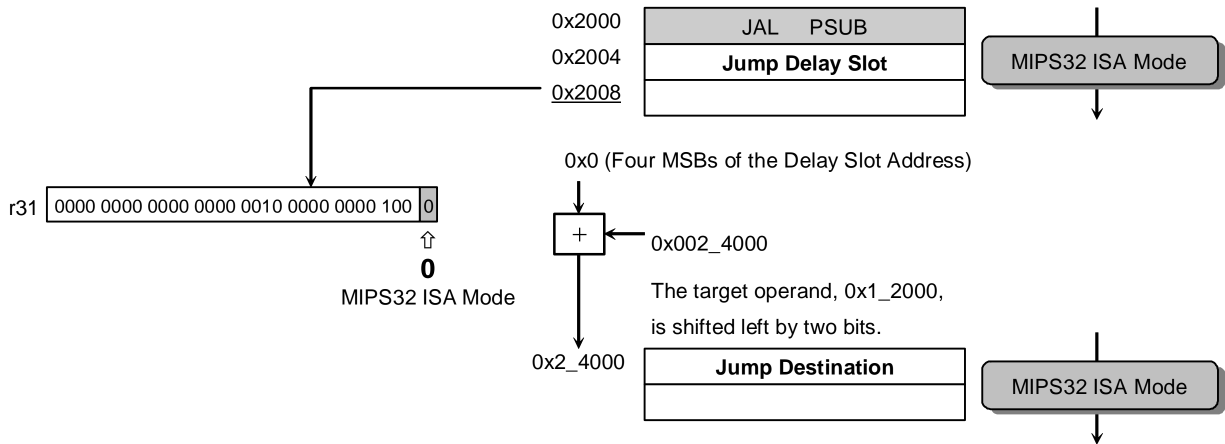
None

Example

```
JAL PSUB
```

Assume that this jump instruction resides at address 0x2000 and that label PSUB points to absolute address 0x2_4000. Then the assembler/linker turns this label into target operand 0x1_2000 (see the figure below).

The processor unconditionally transfers program control to address 0x2_4000. The jump takes effect after the instruction in the jump delay slot is executed. The address of the instruction after the jump delay slot is saved in the link register, r31.


0x2000
JAL PSUB
0x2004
Jump Delay Slot
0x2008
MIPS32 ISA Mode
r31
0000 0000 0000 0000 0010 0000 0000 100
0
0
MIPS32 ISA Mode
0x0 (Four MSBs of the Delay Slot Address)
+
0x002_4000
The target operand, 0x1_2000,
is shifted left by two bits.
0x2_4000
Jump Destination
MIPS32 ISA Mode

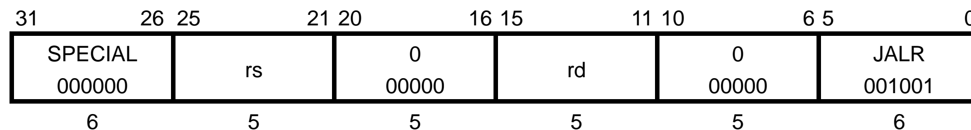
JALR (*rd*,) *rs*

Jump And Link Register

Operation

rd or r31 $\Leftarrow$ pc + 8; pc $\Leftarrow$ rs

Instruction Encoding



Description

The program unconditionally jumps to the address contained in general-purpose register *rs*, with the least-significant bit cleared, with a delay of one instruction (or two pipeline cycles). The least-significant bit of *rs* is interpreted as the ISA mode specifier. The address of the instruction after the jump delay slot (PC+8) is saved in general-purpose register *rd*. If *rd* is omitted, the default is r31 (ra).

Register specifies *rd* and *rs* must not be equal because such an instruction cannot be restarted, with the
contents of *rs* altered by the return address. An exception or interrupt could prevent the completion of a legal instruction in the jump delay slot. If that happens, after the exception handler routine has been executed, processing must restart with the jump instruction.

In 32-bit ISA mode, all instructions must be aligned on word boundaries. Therefore, when jumping to a 32-bit routine, the two low-order bits of the target register (*rs*) must be zero. If the two low-order bits are not zero, an Address Error exception will occur when the processor fetches the instruction at the jump destination.

Exceptions

None

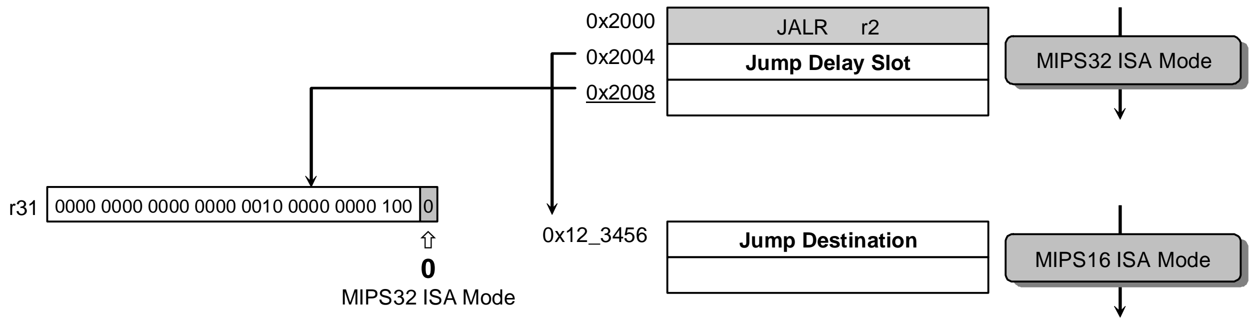
Example

Assume that register r2 contains 0x0012_3457 and that the following jump instruction resides at address 0x0000_2000. Then, executing the instruction:

```
JALR r2
```

transfers program control to address 0x0012_3456, with the least-significant bit of 0x0012_3457

cleared. The jump takes effect after the instruction in the jump delay slot is executed. Since register r2 has the least-significant bit set to 1, the ISA mode bit toggles to 1 after the jump, bringing the processor into 16-bit ISA mode. The return address, 0x0000_2008, is saved in the link register, r31, combined with the ISA mode bit.



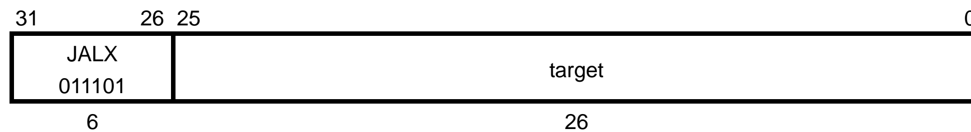
JALX *target*

Jump And Link eXchange

Operation

r31 ⇐ pc + 8; pc[31:1] ⇐ pc[31:28] || *target* || 00; pc[0] ⇐ NOT pc[0]

Instruction Encoding



Description

The program unconditionally jumps to the target address with a delay of one instruction (or two pipeline cycles). The target address is computed relative to the address of the instruction in the jump delay slot (PC+4). The 26-bit *target* is shifted left by two bits and combined with the four most-significant bits of PC+4 to form the target address. The JALX instruction unconditionally toggles the ISA mode bit of the program counter (PC).

The address of the instruction after the jump delay slot (PC+8) is saved in the link register, r31 (ra). The least-significant bit of r31 stores the ISA mode bit that was in effect before the jump.

Exceptions

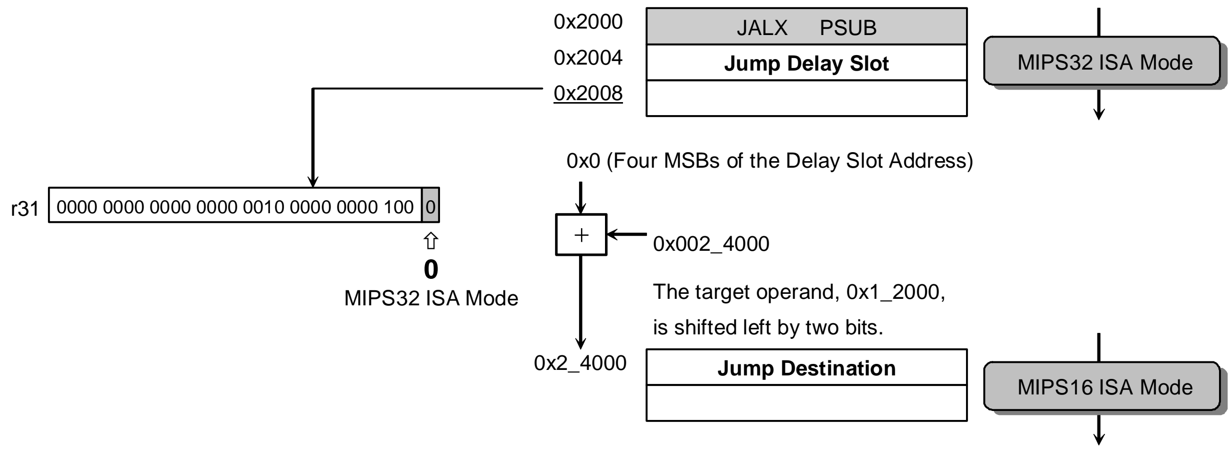
None

Example

```
JALX PSUB
```

Assume that this jump instruction resides at address 0x0000_2000 and that label PSUB points to absolute address 0x2_4000. Then, the assembler/linker turns this label into target operand 0x1_2000 (see the figure below).

The processor unconditionally transfers program control to address 0x2_4000. The jump takes effect after the instruction in the jump delay slot is executed. The ISA mode bit unconditionally toggles, bringing the processor into 16-bit ISA mode. The return address, 0x0000_2008, is saved in the link register, r31, combined with the ISA mode bit.


0x2000
JALX PSUB
0x2004
Jump Delay Slot
0x2008
MIPS32 ISA Mode
0x0 (Four MSBs of the Delay Slot Address)
r31
0000 0000 0000 0000 0010 0000 0000 100 0
0
MIPS32 ISA Mode
+
0x002_4000
The target operand, 0x1_2000,
is shifted left by two bits.
0x2_4000
Jump Destination
MIPS16 ISA Mode

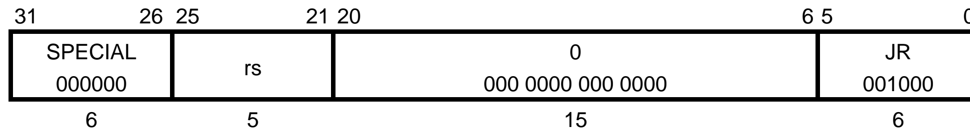
JR *rs*

Jump Register

Operation

pc ⇐ *rs*

Instruction Encoding



Description

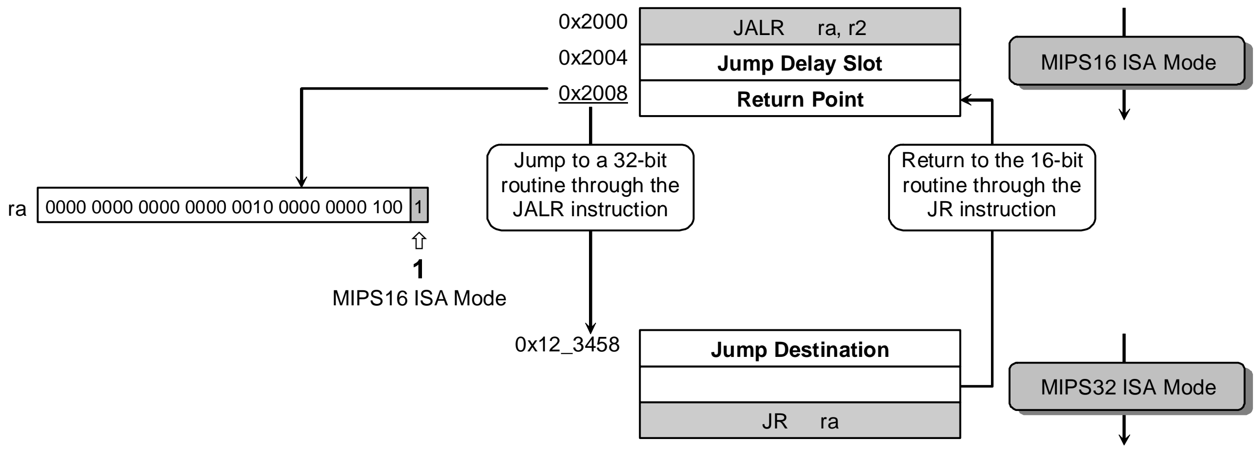
The program unconditionally jumps to the address contained in general-purpose register *rs*, with the least-significant bit cleared, with a delay of one instruction (or two pipeline cycles). The least-significant bit of *rs* is interpreted as the ISA mode specifier.
In 32-bit ISA mode, all instructions must be aligned on word boundaries. Therefore, when jumping to a 32-bit routine, the two low-order bits of the target register (*rs*) must be zero. If the two low-order bits are not zero, an Address Error exception will occur when the processor fetches the instruction at the jump destination.

Exceptions

None

Example

In the following example, the JALR instruction in a 16-bit routine transfers control to a 32-bit routine. At the end of the 32-bit routine, the JR instruction restores the return address into the program counter (PC) from the link register, r31 (ra). Since the JALR instruction saves the ISA mode specifier in the least-significant bit of ra, executing the JR instruction at the end of the 32-bit routine restores it into the PC, causing the processor to revert to 16-bit ISA mode.


0x2000
JALR ra, r2
0x2004
Jump Delay Slot
0x2008
Return Point
MIPS16 ISA Mode
ra
0000 0000 0000 0000 0010 0000 0000 100
1
Jump to a 32-bit routine through the JALR instruction
Return to the 16-bit routine through the JR instruction
1
MIPS16 ISA Mode
0x12_3458
Jump Destination
JR ra
MIPS32 ISA Mode

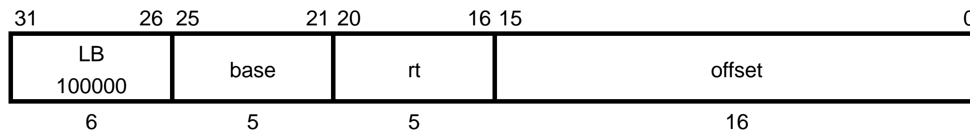
LB *rt*, *offset* (*base*)

Load Byte

Operation

$rt \Leftarrow \{\text{sign-extend}(offset) + (base)\}$

Instruction Encoding



Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The byte in memory addressed by EA is sign-extended and loaded into general-purpose register *rt*.

Exceptions

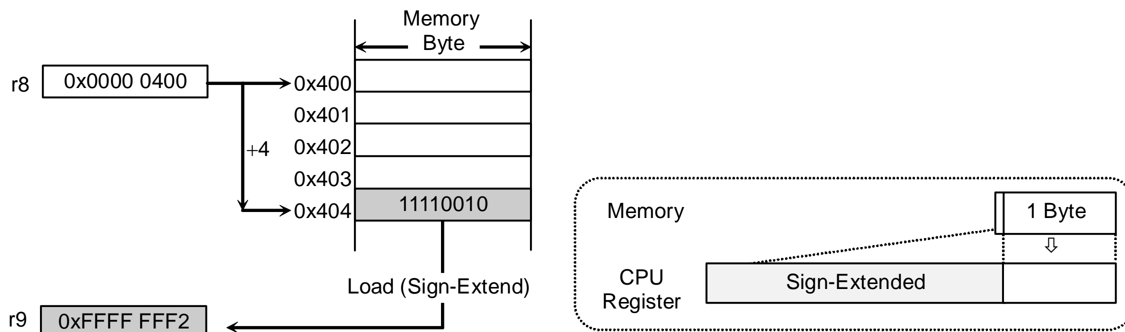
Address Error exception

Example

Assume that register r8 contains 0x0000_0400 and that the memory location at address 0x404 contains 0xF2. Then, executing the instruction:

```
LB r9,4(r8)
```

loads register r9 with 0xFFFF_FFF2.



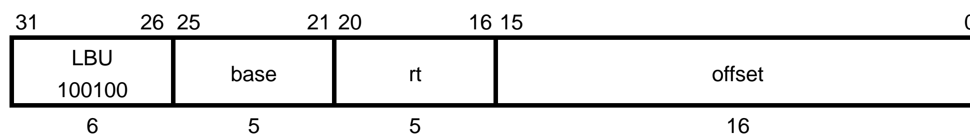
LBU *rt*, *offset* (*base*)

Load Byte Unsigned

Operation

rt ⇐ {sign-extend(*offset*) + (*base*)}

Instruction Encoding



Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The byte in memory addressed by EA is zero-extended and loaded into general-purpose register *rt*.

Exceptions

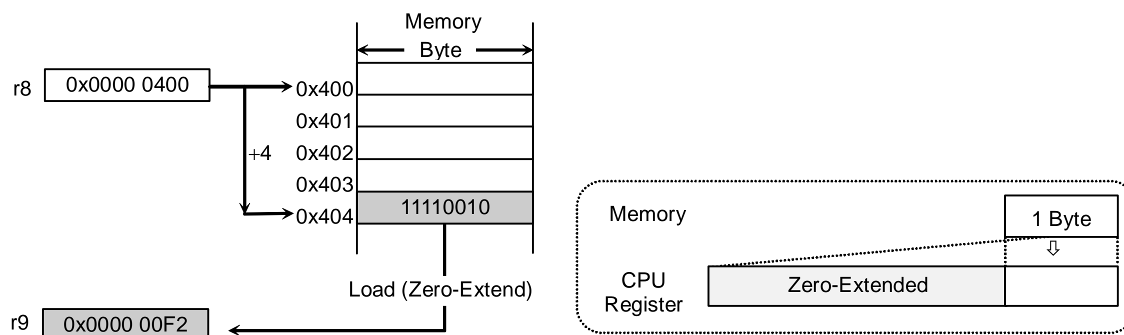
Address Error exception

Example

Assume that register r8 contains 0x0000_0400 and that the memory location at address 0x404 contains 0xF2. Then, executing the instruction:

```
LBU r9,4(r8)
```

loads register r9 with 0x0000_00F2.



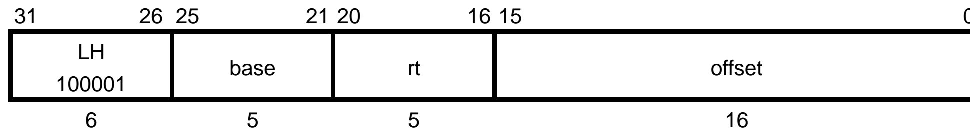
LH *rt*, *offset* (*base*)

Load Halfword

Operation

$rt \Leftarrow \{\text{sign-extend}(offset) + (base)\}$

Instruction Encoding



Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The halfword in memory addressed by EA is sign-extended and loaded into general-purpose register *rt*.

If the least-significant bit of the effective address is not zero (i.e., the effective address is not on a halfword boundary), an Address Error exception occurs.

Exceptions

Address Error exception

Example

Assume that register r8 contains 0x0000_0400 and that the memory locations at addresses 0x404 and 0x405 contain 0xFF and 0x02 respectively. Then, executing the instruction:

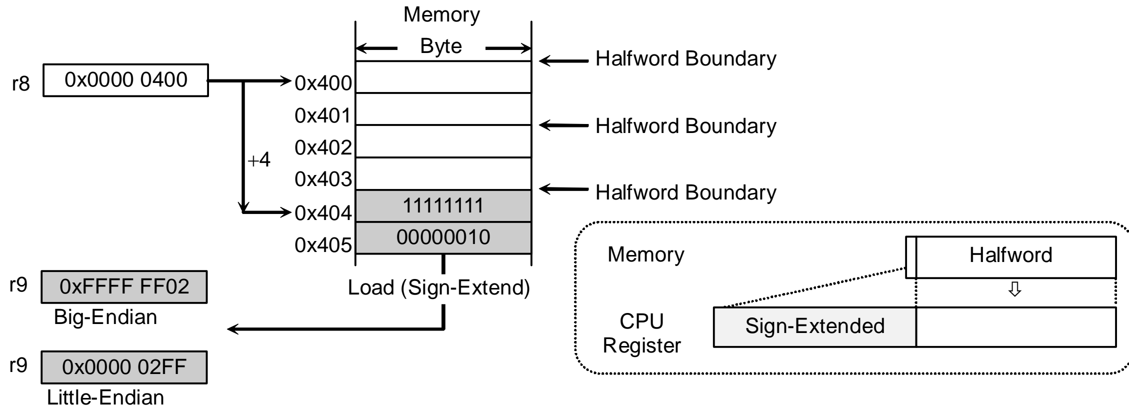
```
LH r9,4(r8)
```

loads register r9 with 0xFFFF_FF02 in big-endian mode and with 0x0000_02FF in little-endian mode.

Executing the instruction:

```
LH r9,3(r8)
```

causes an Address Error exception since 0x403 is not on a halfword boundary.


Memory
Byte
Halfword Boundary
r8
0x0000 0400
0x400
0x401
Halfword Boundary
0x402
+4
0x403
Halfword Boundary
0x404
11111111
0x405
00000010
Memory
Halfword
r9
0xFFFF FF02
Load (Sign-Extend)
Big-Endian
CPU
Register
Sign-Extended
r9
0x0000 02FF
Little-Endian

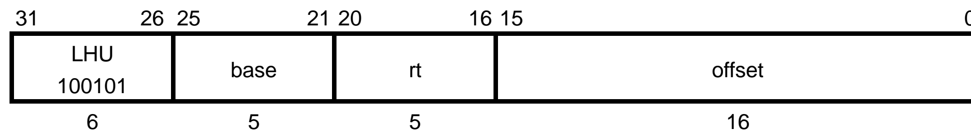
LHU *rt*, *offset* (*base*)

Load Halfword Unsigned

Operation

$rt \Leftarrow \{\text{sign-extend}(offset) + (base)\}$

Instruction Encoding



Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The halfword in memory addressed by EA is zero-extended and loaded into general-purpose register *rt*.

If the least-significant bit of the effective address is not zero (i.e., the effective address is not on a halfword boundary), an Address Error exception occurs.

Exceptions

Address Error exception

Example

Assume that register r8 contains 0x0000_0400 and that the memory locations at addresses 0x404 and 0x405 contain 0xFF and 0x02 respectively. Then, executing the instruction:

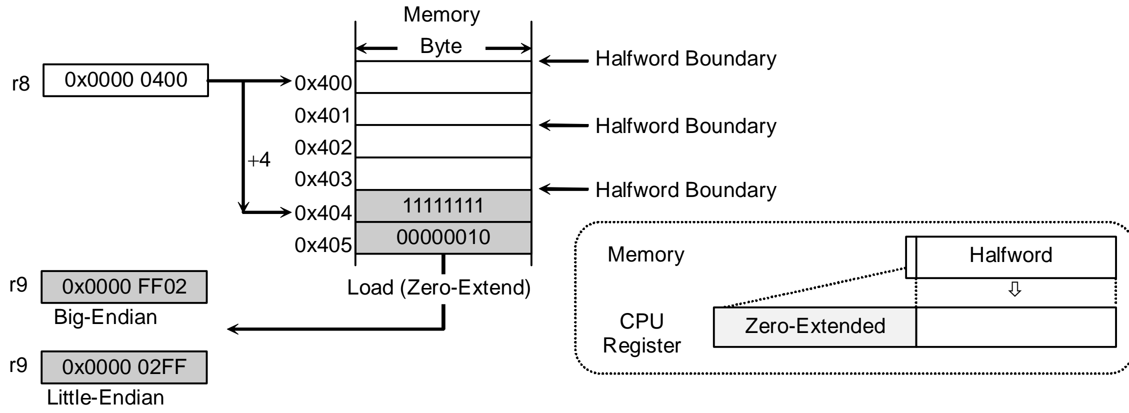
```
LHU r9,4(r8)
```

loads register r9 with 0x0000_FF02 in big-endian mode and with 0x0000_02FF in little-endian mode.

Executing the instruction:

```
LH r9,3(r8)
```

causes an Address Error exception since 0x403 is not on a halfword boundary.


Memory
Byte
Halfword Boundary
r8
0x0000 0400
0x400
0x401
Halfword Boundary
0x402
+4
0x403
Halfword Boundary
0x404
11111111
0x405
00000010
Memory
Halfword
r9
0x0000 FF02
Load (Zero-Extend)
Big-Endian
CPU
Register
Zero-Extended
r9
0x0000 02FF
Little-Endian

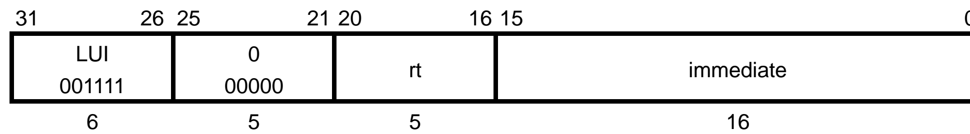
LUI *rt*, *immediate*

Load Upper Immediate

Operation

$rt \Leftarrow immediate \,||\, \text{0x0000}$

Instruction Encoding



Description

The 16-bit *immediate* is shifted left by 16 bits and concatenated to 16 bits of zeros. The result is placed into general-purpose register *rt*.

Exceptions

None

Example

The instruction:

```
LUI r9,0x1234
```

loads register r9 with 0x1234_0000.

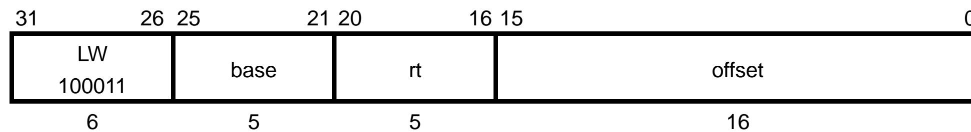
LW *rt*, *offset* (*base*)

Load Word

Operation

$rt \Leftarrow \{\text{sign-extend}(offset) + (base)\}$

Instruction Encoding



Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The word in memory addressed by EA is loaded into general-purpose register *rt*.

If the two low-order bits of the effective address is not zero (i.e., the effective address is not on a word boundary), an Address Error exception occurs.

Exceptions

Address Error exception

Example

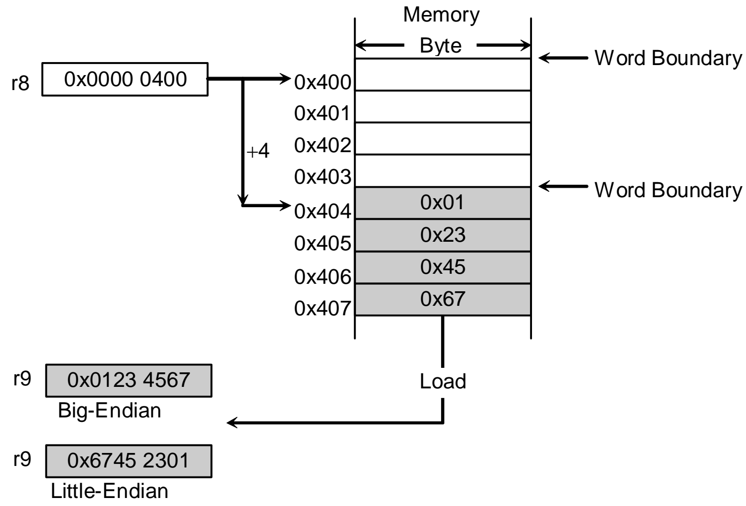
Assume that register r8 contains 0x0000_0400 and that the memory locations at addresses 0x404 to 0x407 contain 0x01, 0x23, 0x45 and 0x67 respectively. Then, executing the instruction:

```
LW r9,4(r8)
```

loads register r9 with 0x0123_4567 in big-endian mode and with 0x6745_2301 in little-endian mode.
Executing the instruction:

```
LW r9,5(r8)
```

causes an Address Error exception since 0x405 is not on a word boundary.


Memory
Byte
Word Boundary
r8
0x0000 0400
0x400
0x401
+4
0x402
0x403
Word Boundary
0x404
0x01
0x405
0x23
0x406
0x45
0x407
0x67
r9
0x0123 4567
Load
Big-Endian
r9
0x6745 2301
Little-Endian

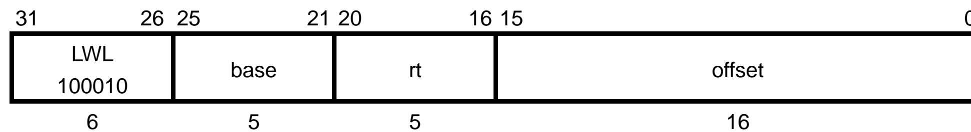
LWL *rt*, *offset* (*base*)

Load Word Left

Operation

$rt \Leftarrow rt$ MERGE {sign-extend(*offset*) + (*base*)}

Instruction Encoding



Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The appropriate high-order part of the word in memory addressed by EA that crosses a natural word boundary is loaded into the left portion of general purpose register *rt*.
No Address Error exception occurs due to misalignment.

An immediately preceding load instruction and the following LWL instruction can specify the same general-purpose register as *rt*. The contents of general-purpose register *rt* is internally bypassed (or forwarded) within the processor so that no NOP instruction is needed between the two instructions.

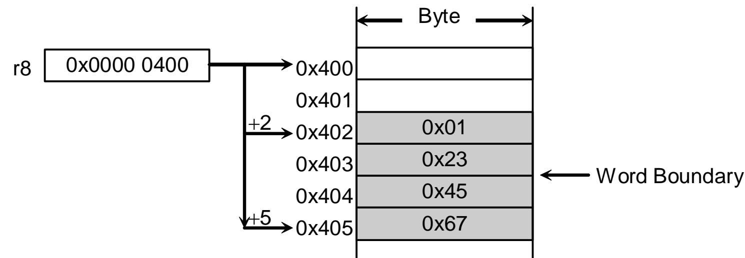
The LWL and LWR instructions are used in combination to load a misaligned word from memory into a general-purpose register.

Exceptions

Address Error exception

Example

Assume that register r8 contains 0x0000_0400 and that the memory locations at addresses 0x402 to 0x405 contains 0x01, 0x23, 0x45 and 0x67 respectively.

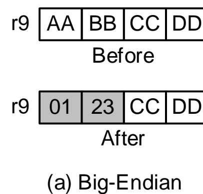


- Big-endian mode

The instruction:

```
LWL r9,2(r8)
```

starts at address 0x402 and loads that byte into the leftmost byte of register r9. Then it loads bytes from memory to r9, going in the higher-address direction, until it reaches a word boundary in memory. The operation of this LWL instruction is as follows.



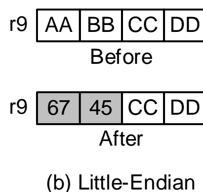
(a) Big-Endian

- Little-endian mode

The instruction:

```
LWL r9,5(r8)
```

starts at address 0x405 and loads that byte into the leftmost byte of register r9. Then it loads bytes from memory to r9, going in the lower-address direction, until it reaches a word boundary in memory. The operation of this LWL instruction is as follows.



(b) Little-Endian

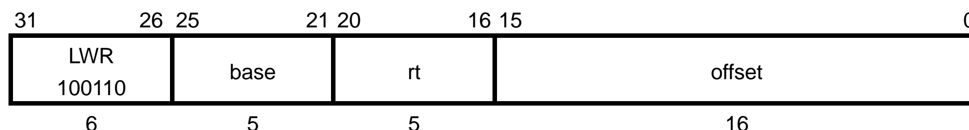
LWR *rt*, *offset* (*base*)

Load Word Right

Operation

$rt \Leftarrow rt$ MERGE {sign-extend(*offset*) + (*base*)}

Instruction Encoding



Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The appropriate low-order part of the word in memory addressed by EA that crosses a natural word boundary is loaded into the right portion of general purpose register *rt*.
No Address Error exception occurs due to misalignment.

An immediately preceding load instruction and the following LWR instruction can specify the same general-purpose register as *rt*. The contents of general-purpose register *rt* is internally bypassed (or forwarded) within the processor so that no NOP instruction is needed between the two instructions.

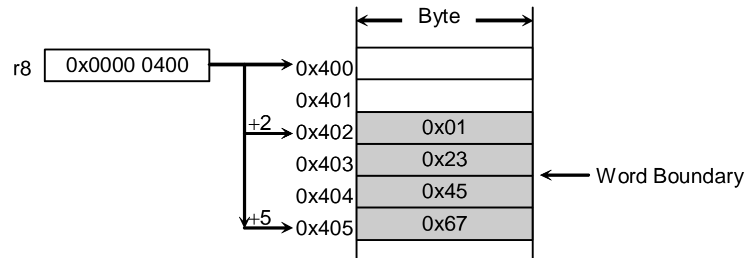
The LWL and LWR instructions are used in combination to load a misaligned word from memory into a general-purpose register.

Exceptions

Address Error exception

Example

Assume that register r8 contains 0x0000_0400 and that the memory locations at addresses 0x402 to 0x405 contains 0x01, 0x23, 0x45 and 0x67 respectively.



- Big-endian mode

The instruction:

```
LWR r9,5(r8)
```

starts at address 0x405 and loads that byte into the rightmost byte of register r9. Then it loads bytes from memory to r9, going in the lower-address direction, until it reaches a word boundary in memory. The operation of this LWR instruction is as follows.

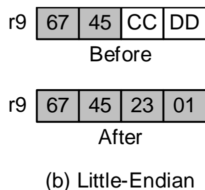
(a) Big-Endian

- Little-endian mode

The instruction:

```
LWR r9,2(r8)
```

starts at address 0x402 and loads that byte into the rightmost byte of register r9. Then it loads bytes from memory to r9, going in the higher-address direction, until it reaches a word boundary in memory. The operation of this LWR instruction is as follows.



(b) Little-Endian

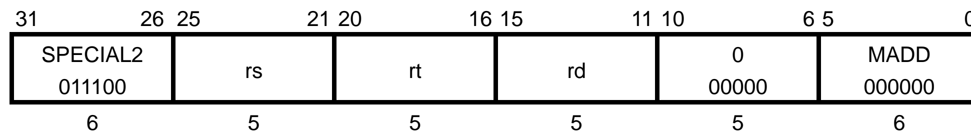
MADD (*rd*,) *rs*, *rt*

Multiply and Add

Operation

HI ⇐ high-order word of (HI || LO) + (*rs* · *rt*);
LO ⇐ low-order word of (HI || LO) + (*rs* · *rt*);
rd ⇐ low-order word of (HI || LO) + (*rs* · *rt*)

Instruction Encoding



Description

The contents of general-purpose register *rs* is multiplied by the contents of general-purpose register *rt*, and then the product is added to the 64-bit, doubleword contents of the HI and LO registers. Both *rs* and *rt* are treated as signed integers. The high-order word of the result is placed into the HI register, and the low-order word of the result is placed into the LO register. If destination register *rd* is specified, the low-order word of the result is also copied into *rd*.

If *rd* is omitted, the default is r0; thus the low-order word of the result is not copied into a general-purpose register.
This instruction never causes an Integer Overflow exception.

Exceptions

None

Example

Assume that the HI and LO registers contain 0x0000_0000 and 0xFFFF_FFFF respectively and that general-purpose registers r2 and r3 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MADD r4,r2,r3
```

evaluates:
0x0000_0000_FFFF_FFFF + (0x0123_4567 · 0x89AB_CDEF)
= 0x0000_0000_FFFF_FFFF + 0xFF79_5E36_C94E_4629
= 0xFF79_5E37_C94E_4628
Hence, the high-order word of the result, 0xFF79_5E37, is placed into the HI register, and the

low-order word of the result, 0xC94E_4628, is placed into the LO and r4 registers.

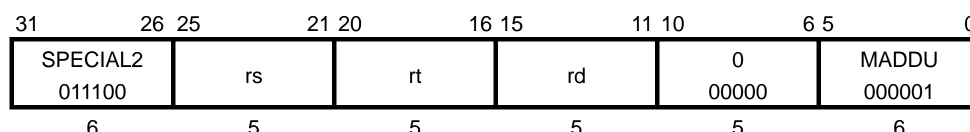
MADDU (*rd*,) *rs*, *rt*

Multiply and Add Unsigned

Operation

$HI \Leftarrow (HI \parallel LO) + (rs \times rt)$ の上位ワード
$LO \Leftarrow (HI \parallel LO) + (rs \times rt)$ の下位ワード
$rd \Leftarrow (HI \parallel LO) + (rs \times rt)$ の下位ワード

Instruction Encoding



Description

The contents of general-purpose register *rs* is multiplied by the contents of general-purpose register *rt*, and then the product is added to the 64-bit, doubleword contents of the HI and LO registers. Both *rs* and *rt* are treated as unsigned integers. The high-order word of the result is placed into the HI register, and the low-order word of the result is placed into the LO register. If destination register *rd* is specified, the low-order word of the result is also copied into *rd*.

If *rd* is omitted, the default is r0; thus the low-order word of the result is not copied into a general-purpose register.
This instruction never causes an Integer Overflow exception.

Exceptions

None

Example

Assume that the HI and LO registers contain 0x_0000_0000 and 0xFFFF_FFFF respectively and that general-purpose registers r2 and r3 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MADDU r4,r2,r3
```

evaluates:
0x0000_0000_FFFF_FFFF + (0x0123_4567 · 0x89AB_CDEF)
= 0x0000_0000_FFFF_FFFF + 0x009C_A39D_C94E_4629
= 0x009C_A39E_C94E_4628
Hence, the high-order word of the result, 0x009C_A39E, is placed into the HI register, and the

low-order word of the result, 0xC94E_4628, is placed into the LO and r4 registers.

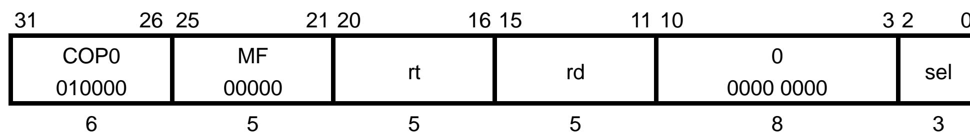
MFC0 *rt*, *rd*

Move From Coprocessor 0

Operation

rt ⇐ coprocessor register *rd* of CP0

Instruction Encoding



Description

The contents of CP0 register *rd* is loaded into general-purpose register *rt*.

Exceptions

Coprocessor Unusable exception

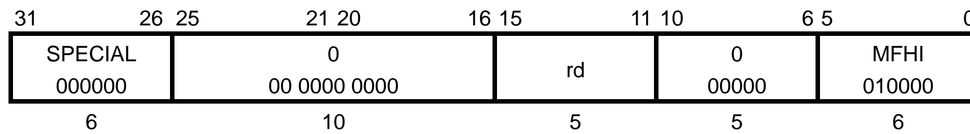
MFHI *rd*

Move From HI

Operation

$rd \Leftarrow \mathrm{HI}$

Instruction Encoding



Description

The contents of HI register is loaded into general-purpose register *rd*.

Exceptions

None

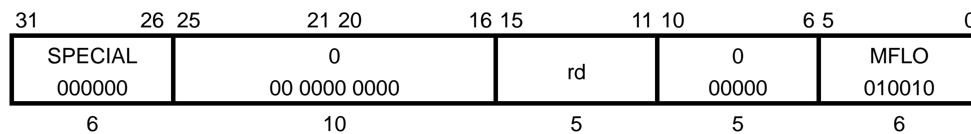
MFLO *rd*

Move From LO

Operation

$rd \Leftarrow \mathrm{LO}$

Instruction Encoding



Description

The contents of the LO register is loaded into general-purpose register *rd*.

Exceptions

None

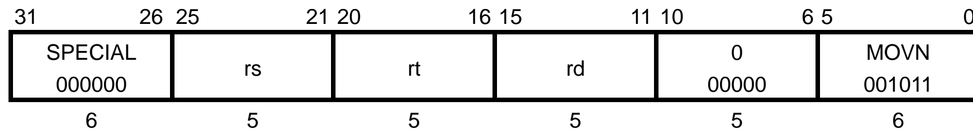
MOVN *rd*, *rs, rt*

Move Conditional on Not Zero

Operation

if $rt \neq 0$ then $rd \Leftarrow rs$

Instruction Encoding



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	MOVN 001011
6	5	5	5	5	6

Description

If the contents of general-purpose register *rt* is not equal to zero, the contents of general-purpose register *rs* is loaded into general-purpose register *rd*.

Exceptions

None

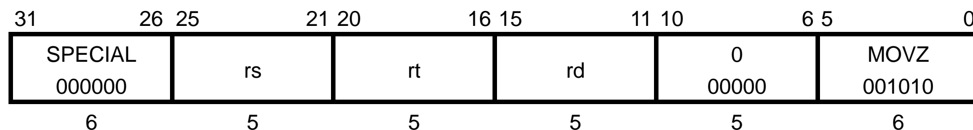
MOVZ *rd*, *rs*, *rt*

Move Conditional on Zero

Operation

if $rt = 0$ then $rd \Leftarrow rs$

Instruction Encoding



Description

If the contents of general-purpose register *rt* is equal to zero, the contents of general-purpose register *rs* is loaded into general-purpose register *rd*.

Exceptions

None

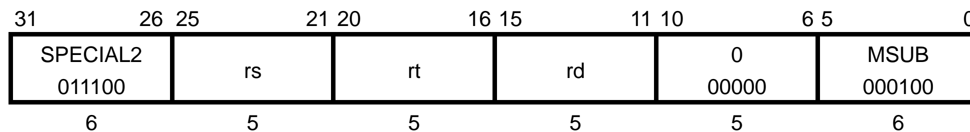
MSUB (*rd*), *rs*, *rt*

Multiply and Subtract

Operation

HI ⇐ high-order word of (HI || LO) – (*rs* · *rt*)
LO ⇐ low-order word of (HI || LO) – (*rs* · *rt*)
rd ⇐ low-order word of (HI || LO) – (*rs* · *rt*)

Instruction Encoding



Description

The contents of general-purpose register *rs* is multiplied by the contents of general-purpose register *rt*, and then the product is subtracted from the 64-bit, doubleword contents of the HI and LO registers. Both *rs* and *rt* are treated as signed integers. The high-order word of the result is placed into the HI register, and the low-order word of the result is placed into the LO register. If destination register *rd* is specified, the low-order word of the result is also copied into *rd*.

If *rd* is omitted, the default is r0; thus the low-order word of the result is not copied into a general-purpose register.
This instruction never causes an Integer Overflow exception.

Exceptions

None

Example

Assume that the HI and LO registers contain 0xFF79_5E37 and 0xC94E_4628 respectively and that general-purpose registers r2 and r3 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MSUB r2,r3
```

evaluates:

0xFF79_5E37_C94E_4628 – (0x0123_4567 · 0x89AB_CDEF)
= 0xFF79_5E37_C94E_4628 – 0xFF79_5E36_C94E_4629
= 0x0000_0000_FFFF_FFFF

Hence, the high-order word of the result, 0x0000_0000, is placed into the HI register, and the low-order word of the result, 0xFFFF_FFFF, is placed into the LO register.

MSUBU (*rd*), *rs*, *rt*

Multiply and Subtract Unsigned

Operation

HI ⇐ high-order word of (HI || LO) – (*rs* · *rt*)
LO ⇐ low-order word of (HI || LO) – (*rs* · *rt*)
rd ⇐ low-order word of (HI || LO) – (*rs* · *rt*)

Instruction Encoding

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL2 011100	rs	rt	rd	0 00000	MSUBU 000101
6	5	5	5	5	6

Description

The contents of general-purpose register *rs* is multiplied by the contents of general-purpose register *rt*, and then the product is subtracted from the 64-bit, doubleword contents of the HI and LO registers. Both *rs* and *rt* are treated as unsigned integers. The high-order word of the result is placed into the HI register, and the low-order word of the result is placed into the LO register. If destination register *rd* is specified, the low-order word of the result is also copied into *rd*.

If *rd* is omitted, the default is r0; thus the low-order word of the result is not copied into a general-purpose register.
This instruction never causes an Integer Overflow exception.

Exceptions

None

Example

Assume that the HI and LO registers contain 0x009C_A39E and 0xC94E_4628 respectively and that general-purpose registers r2 and r3 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MSUBU r2,r3
```

evaluates:

0x009C_A39E_C94E_4628 – (0x0123_4567 · 0x89AB_CDEF)
= 0x009C_A39E_C94E_4628 – 0x009C_A39D_C94E_4629
= 0x0000_0000_FFFF_FFFF

Hence, the high-order word of the result, 0x0000_0000, is placed into the HI register, and the low-order word of the result, 0xFFFF_FFFF, is placed into the LO register.

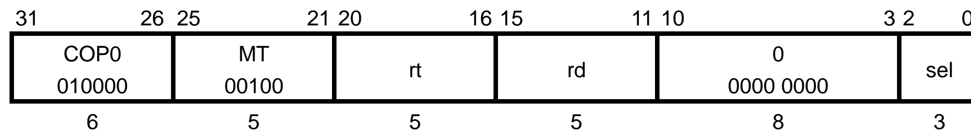
MTC0 *rt*, *rd*

Move To Coprocessor 0

Operation

Coprocessor register *rd* of CP0 ⇐ *rt*

Instruction Encoding



Description

The contents of general-purpose register *rt* is loaded into CP0 register *rd*.

Once the MTC0 instruction writes to the Status, EPC or ErrorEPC register, at least two instructions must be executed before the ERET instruction. Otherwise, the operation is undefined.

Likewise, once the MTC0 instruction writes to the DEPC register, at least two instructions must be executed before the DERET instruction. Otherwise, the operation is undefined.

Because this instruction may alter the state of the virtual address translation system, the operation of load and store instructions immediately before and after this instruction is undefined.

The MTC0 instruction that modifies the contents of the SSCR register must be followed by two NOPs.

Exceptions

Coprocessor Unusable exception

MTHI *rs*

Move To HI

Operation

HI ⇐ *rs*

Instruction Encoding

31 ... 26	25 ... 21	20 ... 6	5 ... 0
SPECIAL 000000	rs	0 000 0000 0000 0000	MTHI 010001
6	5	15	6

Description

The contents of general-purpose register *rs* is loaded into the HI register.

Exceptions

None

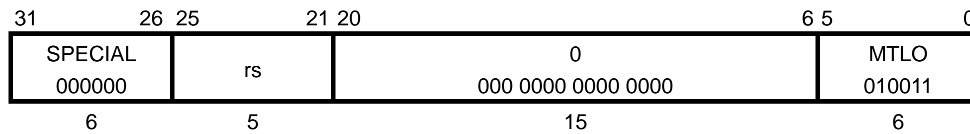
MTLO *rs*

Move To LO

Operation

LO ⇐ *rs*

Instruction Encoding



Description

The contents of general-purpose register *rs* is loaded into the LO register.

Exceptions

None

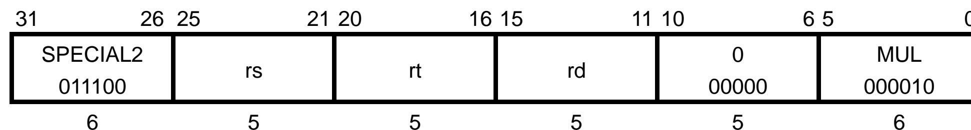
MUL *rd*, *rs*, *rt*

Multiply

Operation

$rd \Leftarrow$ low-order word of $(rs \cdot rt)$

Instruction Encoding



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL2 011100	rs	rt	rd	0 00000	MUL 000010
6	5	5	5	5	6

Description

The contents of general-purpose register *rs* is multiplied by the contents of general-purpose register *rt*. Both *rs* and *rt* are treated as signed integers. The low-order word of the result is placed into general-purpose register *rd*. The contents of the HI and LO registers become undefined.

This instruction never causes an Integer Overflow exception.

Exceptions

None

Example

Assume that general-purpose registers r2 and r3 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MUL r4,r2,r3
```

evaluates:
(0x0123_4567 · 0x89AB_CDEF)
= 0xFF79_5E36_C94E_4629
Hence, the low-order word of the result, 0xC94E_4629, is placed into the r4 register.

MULT (*rd*,) *rs*, *rt*

Multiply

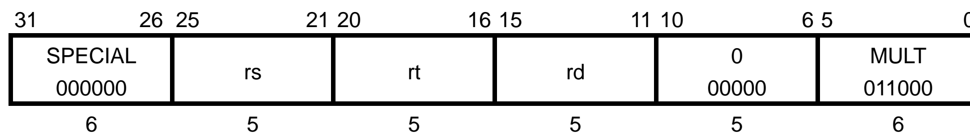
Operation

HI ⇐ high-order word of (*rs* · *rt*);

LO ⇐ low-order word of (*rs* · *rt*);

rd ⇐ low-order word of (*rs* · *rt*)

Instruction Encoding



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	MULT 011000
6	5	5	5	5	6

Description

The contents of general-purpose register *rs* is multiplied by the contents of general-purpose register *rt*. Both *rs* and *rt* are treated as signed integers. The high-order word of the result is placed into the HI register, and the low-order word of the result is placed into the LO register. If destination register *rd* is specified, the low-order word of the result is also copied into *rd*.

If *rd* is omitted, the default is r0; thus the low-order word of the result is not copied into a general-purpose register.
This instruction never causes an Integer Overflow exception.

Exceptions

None

Example

Assume that general-purpose registers r2 and r3 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MULT r4,r2,r3
```

evaluates:
(0x0123_4567 · 0x89AB_CDEF)
= 0xFF79_5E36_C94E_4629
Hence, the high-order word of the result, 0xFF79_5E36, is placed into the HI register, and the low-order word of the result, 0xC94E_4629, is placed into the LO and r4 registers.

MULTU (*rd*,) *rs*, *rt*

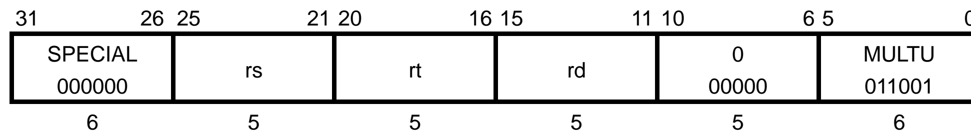
Multiply Unsigned

Operation

HI ⇐ high-order word of (*rs* · *rt*);

LO ⇐ low-order word of (*rs* · *rt*);
rd ⇐ low-order word of (*rs* · *rt*)

Instruction Encoding



Description

The contents of general-purpose register *rs* is multiplied by the contents of general-purpose register *rt*. Both *rs* and *rt* are treated as unsigned integers. The high-order word of the result is placed into the HI register, and the low-order word of the result is placed into the LO register. If destination register *rd* is specified, the low-order word of the result is also copied into *rd*.

If *rd* is omitted, the default is r0; thus the low-order word of the result is not copied into a general-purpose register.
This instruction never causes an Integer Overflow exception.

Exceptions

None

Example

Assume that general-purpose registers r2 and r3 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MULTU r4,r2,r3
```

(0x0123_4567 · 0x89AB_CDEF)
= 0x009C_A39D_C94E_4629
Hence, the high-order word of the result, 0x009C_A39D, is placed into the HI register, and the low-order word of the result, 0xC94E_4629, is placed into the LO and r4 registers.

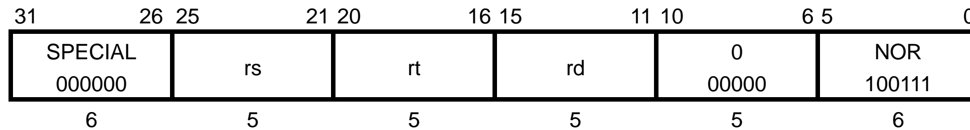
NOR *rd*, *rs*, *rt*

NOR

Operation

$rd \Leftarrow rs$ NOR rt

Instruction Encoding



Description

The contents of general-purpose register *rs* is NORed with the contents of general-purpose register *rt*, and the result is placed into general-purpose register *rd*.

Exceptions

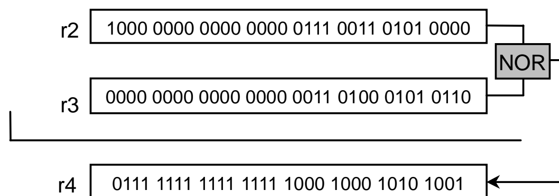
None

Example

Assume that registers r2 and r3 contain 0x8000_7350 and 0x0000_3456 respectively. Then, the instruction:

```
NOR r4,r2,r3
```

performs the logical NOR between r2 and r3 and puts the result (0x7FFF_88A9) in r4, as shown below.



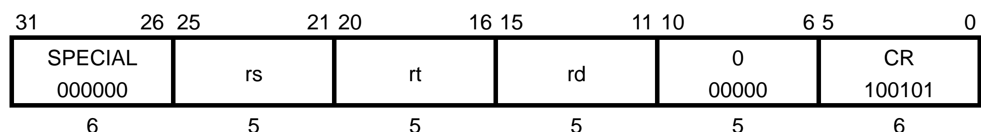
OR *rd*, *rs*, *rt*

OR

Operation

$rd \Leftarrow rs$ OR rt

Instruction Encoding



Description

The contents of general-purpose register *rs* is ORed with the contents of general-purpose register *rt*, and the result is placed into general-purpose register *rd*.

Exceptions

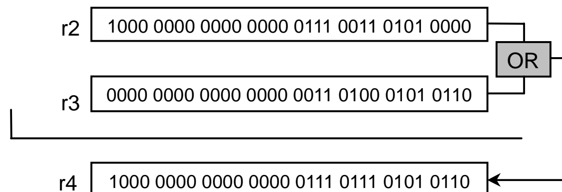
None

Example

Assume that registers r2 and r3 contain 0x8000_7350 and 0x0000_3456 respectively. Then, the instruction:

```
OR r4,r2,r3
```

performs the logical OR between r2 and r3 and puts the result (0x8000_7756) in r4, as shown below.



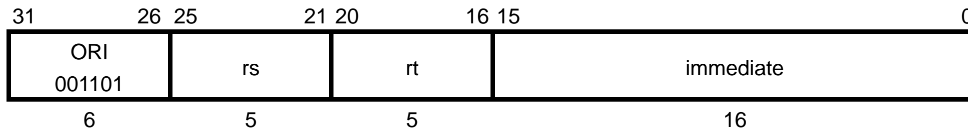
ORI *rt*, *rs*, *immediate*

OR Immediate

Operation

$rt \Leftarrow rs$ OR $(0^{16} \parallel immediate_{15..0})$

Instruction Encoding



Description

The 16-bit *immediate* is zero-extended and ORed with the contents of general-purpose register *rs*. The result is placed into general-purpose register *rt*.

The *immediate* field is 16 bits in length. If the *immediate* size is larger than that, you need to put it in a general-purpose register and use the OR instruction (see Section 3.3.2, *32-Bit Constants*).

Exceptions

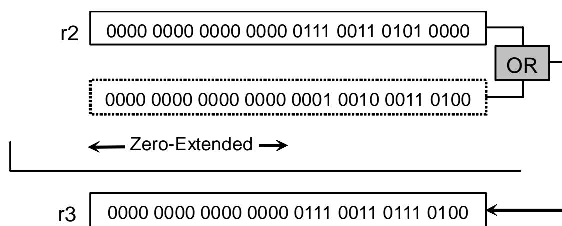
None

Example

Assume that register r2 contains 0x0000_7350. Then, the instruction:

```
ORI r3,r2,0x1234
```

performs the logical OR between 0x0000_7350 and 0x0000_1234 and puts the result (0x0000_7374) in r3, as shown below.



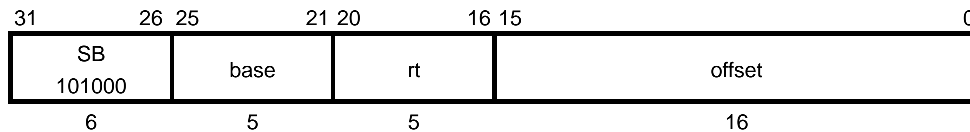
SB *rt*, *offset* (*base*)

Store Byte

Operation

$rt \Rightarrow \{\text{sign-extend}(offset) + (base)\}$

Instruction Encoding



Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register base to form an effective address (EA). The least-significant byte in general-purpose register *rt* is stored at the memory location addressed by EA.

The three high-order bytes in *rt* are simply ignored; so there is no distinction between signed and unsigned stores.

Exceptions

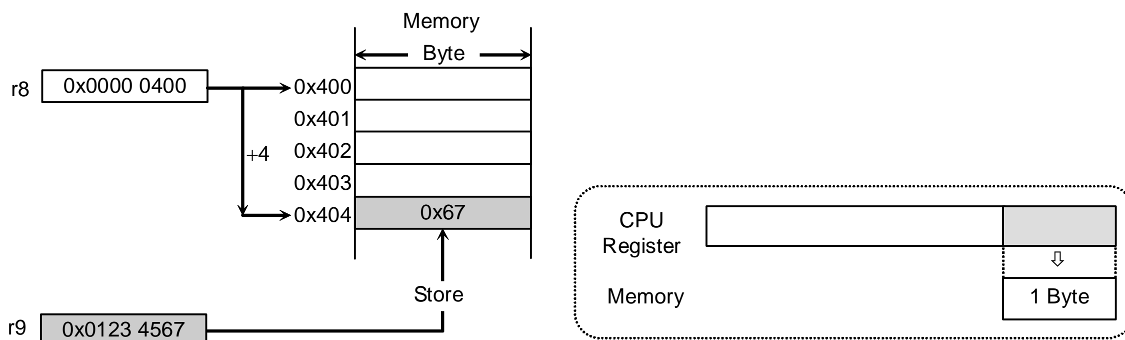
Address Error exception

Example

Assume that registers r8 and r9 contain 0x0000_0400 and 0x0123_4567 respectively. Then, executing the instruction:

```
SB r9,4(r8)
```

stores 0x67 to the memory location at address 0x404.



SDBBP *code*

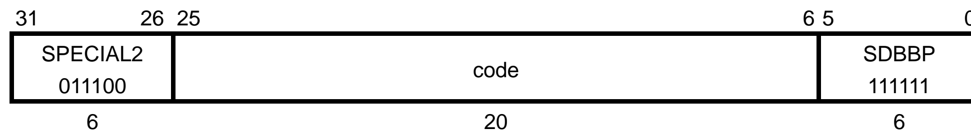
EJTAG

Software Debug Breakpoint Exception

Operation

Software debug breakpoint exception

Instruction Encoding



Description

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

The *code* field in the SDBBP instruction is available for use as software parameters to pass additional information. The exception handler can retrieve it by loading the contents of the memory word containing the instruction. See Section 9.3, *Debug Exceptions*, for details.

The SDBBP instruction may not be used within the user’s program; it is intended for use by development tools. Executing the SDBBP instruction on a device without EJTAG causes a Reserved Instruction exception.

Exceptions

Debug Breakpoint exception
Reserved Instruction exception

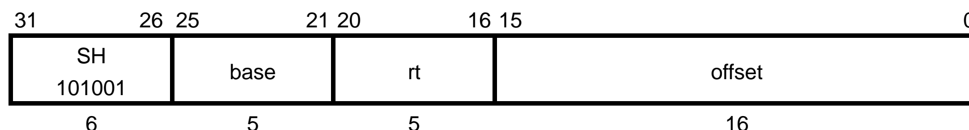
SH *rt*, *offset* (*base*)

Store Halfword

Operation

$rt \Rightarrow \{\text{sign-extend}(offset) + (base)\}$

Instruction Encoding



Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The least-significant halfword in general-purpose register *rt* is stored at the memory location addressed by EA.

The higher-order halfword in *rt* is simply ignored; so there is no distinction between signed and unsigned stores.

If the least-significant bit of the effective address is not zero (i.e., the effective address is not on a halfword boundary), an Address Error exception occurs.

Exceptions

Address Error exception

Example

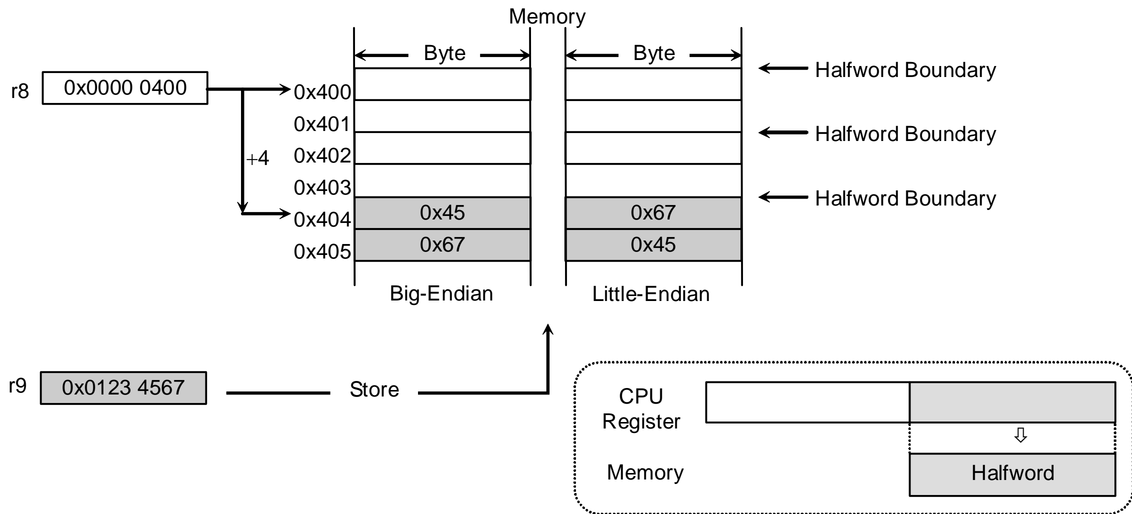
Assume that registers r8 and r9 contain 0x0000_0400 and 0x0123_4567 respectively. In big-endian mode, executing the instruction:

```
SH r9,4(r8)
```

stores 0x45 and 0x67 to the memory locations at addresses 0x404 and 0x405 respectively. In little-endian mode, this instruction stores 0x67 and 0x45 to the memory locations at addresses 0x404 and 0x405 respectively.
Executing the instruction:

```
SH r9,3(r8)
```

causes an Address Error exception since 0x403 is not on a halfword boundary.


Memory
Byte
Byte
r8
0x0000 0400
+4
0x400
0x401
0x402
0x403
0x404
0x405
0x45
0x67
0x67
0x45
Halfword Boundary
Halfword Boundary
Halfword Boundary
Big-Endian
Little-Endian
r9
0x0123 4567
Store
CPU Register
Memory
Halfword

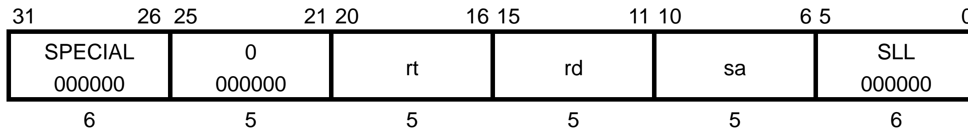
SLL *rd*, *rt*, *sa*

Shift Left Logical

Operation

$rd \Leftarrow rt << sa$

Instruction Encoding



Description

The contents of general-purpose register *rt* is shifted left by *sa* bits. Zeros are supplied to the vacated positions on the right. The result is placed into general-purpose register *rd*.

Exceptions

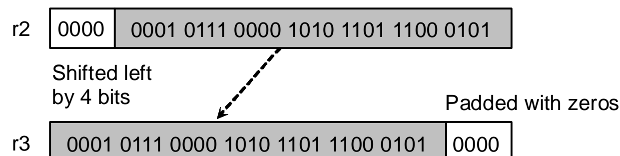
None

Example

Assume that register r2 contains 0x2170_ADC5. Then, executing the instruction:

```
SLL r3,r2,4
```

places 0x170A_DC50 in register r3, as shown below.



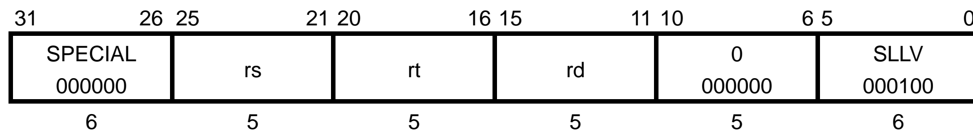
SLLV *rd*, *rt*, *rs*

Shift Left Logical Variable

Operation

$rd \Leftarrow rt \ll 5$ LSBs of rs

Instruction Encoding



Description

The contents of general-purpose register *rt* is shifted left the number of bits specified by the five least-significant bits of general-purpose register *rs*. Zeros are supplied to the vacated positions on the right. The result is placed into general-purpose register *rd*.

Exceptions

None

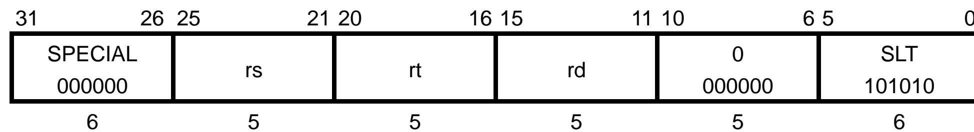
SLT *rd*, *rs*, *rt*

Set On Less Than

Operation

if $rs < rt$ then $rd \Leftarrow 1$; else $rd \Leftarrow 0$

Instruction Encoding



Description

The contents of general-purpose register *rs* is compared to the contents of general-purpose register *rt*. Both *rs* and *rt* are treated as signed integers. If *rs* is less than *rt*, general-purpose register *rd* is set to one. Otherwise, *rd* is set to zero.

No Integer Overflow exception occurs under any circumstances. The comparison is valid even if the subtraction performed for comparison results in overflow.

Exceptions

None

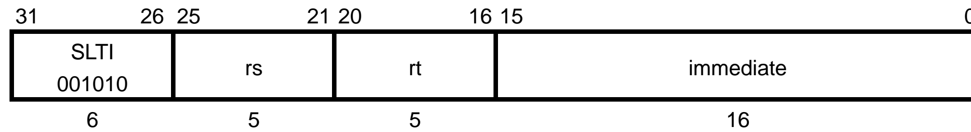
SLTI *rt*, *rs*, *immediate*

Set On Less Than Immediate

Operation

if $rs < ((immediate_{15})^{16} \parallel immediate_{15..0})$ then $rt \Leftarrow 1$; else $rt \Leftarrow 0$

Instruction Encoding



31–26	25–21	20–16	15–0
SLTI 001010	rs	rt	immediate
6	5	5	16

Description

The 16-bit *immediate* is sign-extended and compared to the contents of general-purpose register *rs*. The *immediate* and *rs* are compared as signed integers. If *rs* is less than *immediate*, general-purpose register *rt* is set to one. Otherwise, *rt* is set to zero.

No Integer Overflow exception occurs under any circumstances. The comparison is valid even if the subtraction performed for comparison results in overflow.

With the 16-bit *immediate*, the immediate range is -32768 to +32767. If a number is outside this range, you need to put it in a general-purpose register and use the SLT instruction (see Section 3.3.2, *32-Bit Constants*).

Exceptions

None

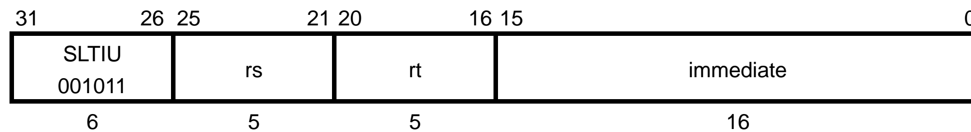
SLTIU *rt*, *rs*, *immediate*

Set On Less Than Immediate Unsigned

Operation

if $(0 \parallel rs) < ((immediate_{15})^{17} \parallel immediate_{15..0})$ then $rt \Leftarrow 1$; else $rt \Leftarrow 0$

Instruction Encoding



Description

The 16-bit immediate is *sign*-extended and compared to the contents of general-purpose register *rs*. The *immediate* and *rs* are compared as unsigned integers. If *rs* is less than *immediate*, general-purpose register *rt* is set to one. Otherwise, *rt* is set to zero.
No Integer Overflow exception occurs under any circumstances. The comparison is valid even if the subtraction performed for comparison results in overflow.

The *immediate* field is 16 bits in length. If a number is outside this range, you need to put it in a general-purpose register and use the SLTU instruction (see Section 3.3.2, *32-Bit Constants*).

Exceptions

None

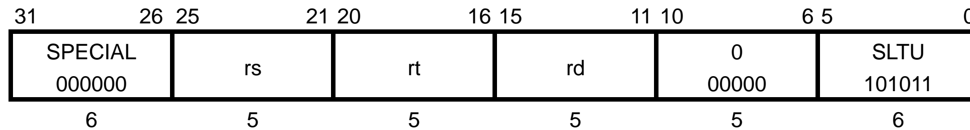
SLTU *rd*, *rs*, *rt*

Set On Less Than Unsigned

Operation

if (0 || *rs*) < (0 || *rt*) then *rd* ⇐ 1; else *rd* ⇐ 0

Instruction Encoding



Description

The contents of general-purpose register *rs* is compared to the contents of general-purpose register *rt*. Both *rs* and *rt* are treated as unsigned integers. If *rs* is less than *rt*, general-purpose register *rd* is set to one. Otherwise, *rd* is set to zero.

No Integer Overflow exception occurs under any circumstances. The comparison is valid even if the subtraction performed for comparison results in overflow.

Exceptions

None

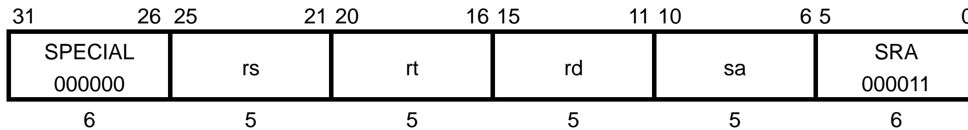
SRA *rd*, *rt*, *sa*

Shift Right Arithmetic

Operand

$rd \Leftarrow rt >> sa$

Instruction Encoding



Description

The contents of general-purpose register *rt* is shifted right by *sa* bits. The sign bit is copied to the vacated positions on the left. The result is placed into general-purpose register *rd*.

Exceptions

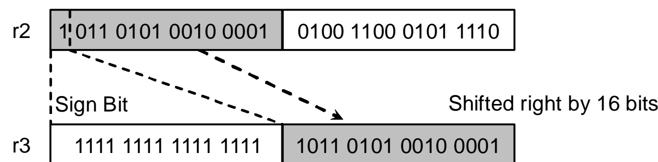
None

Example

Assume that register r2 contains 0xB521_4C5E. Then, executing the instruction:

```
SRA r3,r2,16
```

places 0xFFFF_B521 into r3, as shown below.



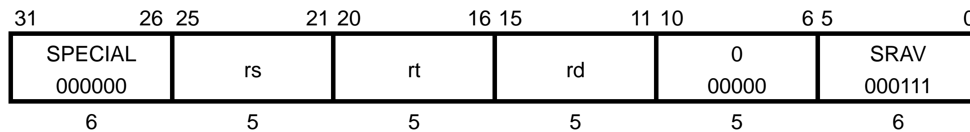
SRAV *rd*, *rt*, *rs*

Shift Right Arithmetic Variable

Operation

rd ⇐ *rt* >> 5 LSBs of *rs*

Instruction Encoding



Description

The contents of general-purpose register *rt* is shifted right the number of bits specified by the five least-significant bits of general-purpose register *rs*. The sign bit is copied to the vacated positions on the left. The result is placed into general-purpose register *rd*.

Exceptions

None

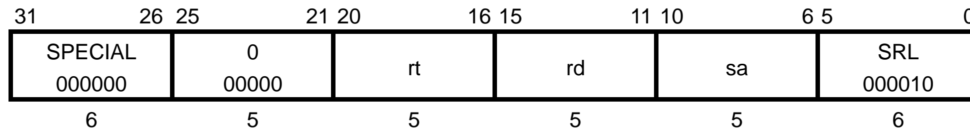
SRL *rd*, *rt*, *sa*

Shift Right Logical

Operation

$rd \Leftarrow rt >> sa$

Instruction Encoding



Description

The contents of general-purpose register *rt* is shifted left by *sa* bits. Zeros are supplied to the vacated positions on the left. The result is placed into general-purpose register *rd*.

Exceptions

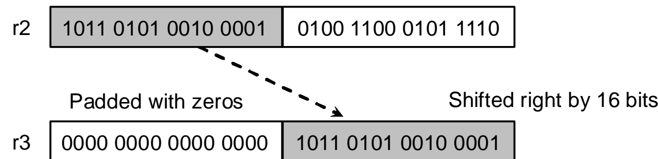
None

Example

Assume that register r2 contains 0xB521_4C5E. Then, executing the instruction:

```
SRL r3,r2,16
```

places 0x0000_B521 in register r3, as shown below.



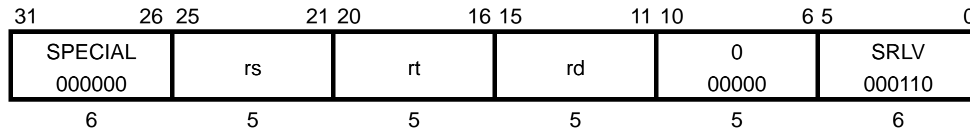
SRLV *rd*, *rt*, *rs*

Shift Right Logical Variable

Operation

rd ⇐ *rt* >> 5 LSBs of *rs*

Instruction Encoding



Description

The contents of general-purpose register *rt* (32 bits) is shifted right the number of bits specified by the five least-significant bits of general-purpose register *rs*. Zeros are supplied to the vacated positions on the left. The result is placed into general-purpose register *rd*.

Exceptions

None

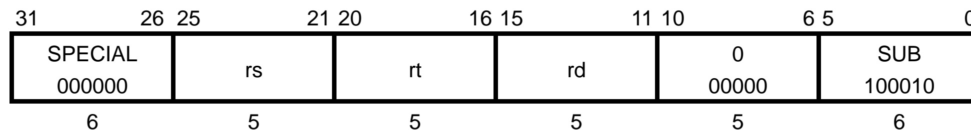
SUB *rd*, *rs*, *rt*

Subtract

Operation

$rd \Leftarrow rs - rt$

Instruction Encoding



Description

The contents of general-purpose register *rt* is subtracted from the contents of general-purpose register *rs*. Both *rs* and *rt* are treated as signed integers. The remainder is placed into general-purpose register *rd*.

An Integer Overflow exception is taken on 2's-complement overflow, which occurs if the signs of the operands are not the same and the sign of the remainder is not the same as the sign of the minuend (*rs*). The destination register (*rd*) is not altered when an Integer Overflow exception occurs.

Exceptions

Interger Overflow exception

Example

1. Assume that registers r2 and r3 contain 0x7654_3210 and 0x5000_0000 respectively. Then, executing the instruction:

```
SUB r4,r2,r3
```

places the remainder (0x2654_3210) into r4.

2. Assume that registers r2 and r3 contain 0x7FFF_FFFF and 0x8FFF_FFFF respectively. Then, the subtraction of r3 from r2 gives the result 0xF000_0000. So, the signs of r2 and r3 are different, and the signs of r2 and the remainder are also different. This indicates a 2'scomplement overflow. Thus executing the instruction:

```
SUB r4,r2,r3
```

causes an Integer Overflow exception. Register r4 is not modified as a result of this instruction.

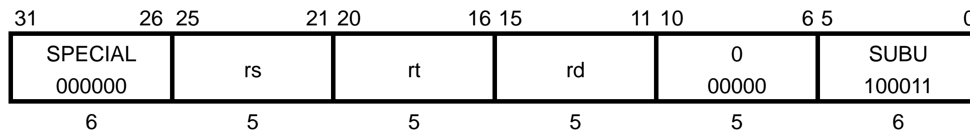
SUBU *rd*, *rs*, *rt*

Subtract Unsigned

Operation

$rd \Leftarrow rs - rt$

Instruction Encoding



Description

The contents of general-purpose register *rt* is subtracted from the contents of general-purpose register *rs*. The remainder is placed into general-purpose register *rd*.

The only difference between this instruction and the SUB instruction is that this instruction never causes an Integer Overflow exception.

Exceptions

None

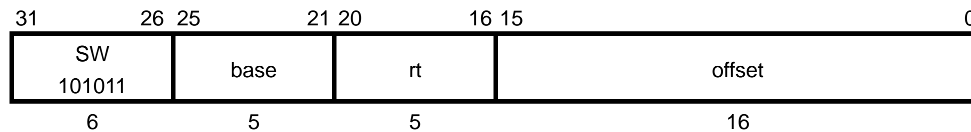
SW *rt*, *offset* (*base*)

Store Word

Operation

$rt \Rightarrow \{\text{sign-extend}(offset) + (base)\}$

Instruction Encoding



Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The contents of general-purpose register *rt* is stored at the memory location addressed by EA.

If the two least-significant bits of the effective address is not zero (i.e., the effective address is not on a word boundary), an Address Error exception occurs.

Exceptions

Address Error exception

Example

Assume that registers r8 and r9 contain 0x0000_0400 and 0x0123_4567 respectively. In big-endian mode, executing the instruction:

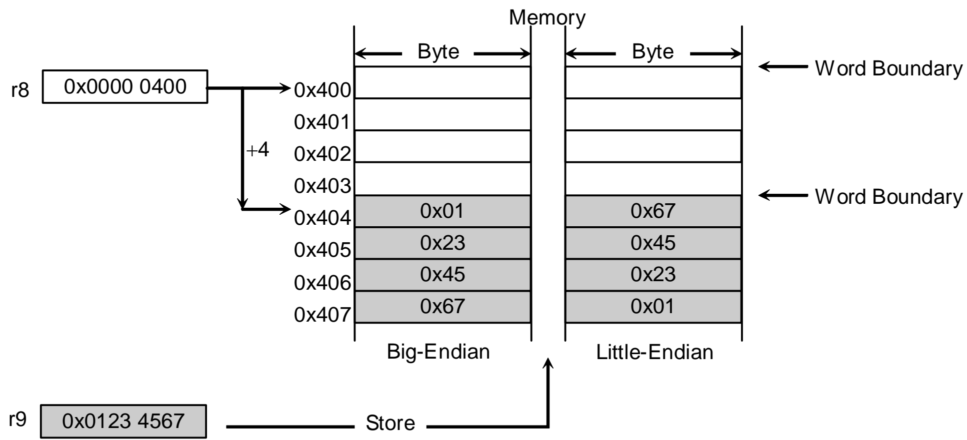
```
SW r9,4(r8)
```

stores 0x01, 0x23, 0x45 and 0x67 to the memory locations at addresses 0x404 to 0x407 respectively. In little-endian mode, this instruction stores 0x67, 0x45, 0x23 and 0x01 to the memory locations at addresses 0x404 to 0x407 respectively.

Executing the instruction:

```
SW r9,5(r8)
```

causes an Address Error exception since 0x405 is not on a word boundary.


Memory
Byte
Byte
Word Boundary
Word Boundary
r8
0x0000 0400
+4
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
0x67
0x45
0x23
0x01
Big-Endian
Little-Endian
r9
0x0123 4567
Store

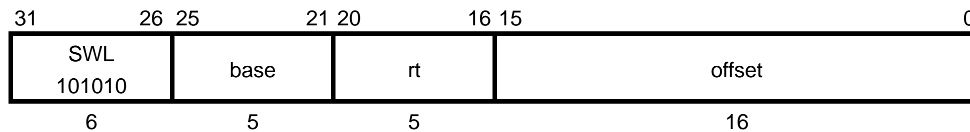
SWL *rt*, *offset* (*base*)

Store Word Left

Operation

$rt \Rightarrow \{\text{sign-extend}(offset) + (base)\}$

Instruction Encoding



Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The left portion of general-purpose register *rt* is stored into the appropriate high-order part of the word at the memory locations addressed by EA that cross a natural word boundary.

No Address Error exception occurs due to misalignment.
The SWL and SWR instructions are used in combination to store a word into memory locations that are not on a natural word boundary.

Exceptions

Address Error exception

Example

Assume that registers r8 and r9 contain 0x0000_0400 and 00123_4567 respectively.

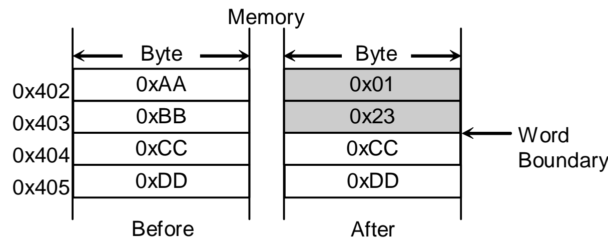
r9 0x0123 4567

- Big-endian mode

The instruction:

```
SWL r9,2(r8)
```

starts at the leftmost byte in register r9 and stores that byte at address 0x0402. Then it stores bytes in register r9, going in the higher-address direction, until it reaches a word boundary in memory. The operation of this SWL instruction is as follows.



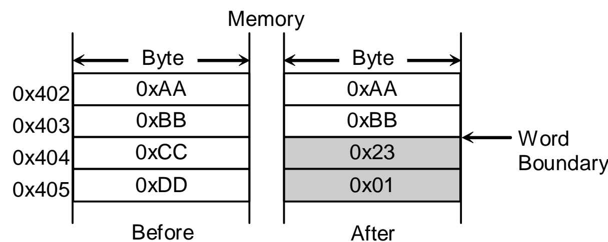
(a) Big-Endian

- Little-endian mode

The instruction:

```
SWL r9,5(r8)
```

starts at the leftmost byte in register r9 and stores that byte at address 0x0405. Then it stores bytes in register r9, going in the lower-address direction, until it reaches a word boundary in memory. The operation of this SWL instruction is as follows.



(b) Little-Endian

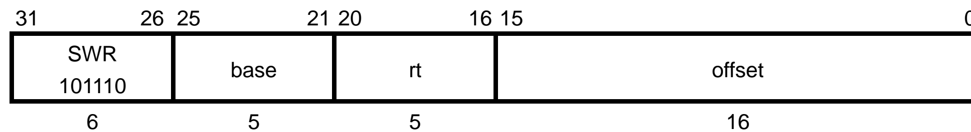
SWR *rt*, *offset* (*base*)

Store Word Right

Operation

$rt \Rightarrow \{\text{sign-extend}(offset) + (base)\}$

Instruction Encoding



Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The right-portion of general-purpose register *rt* is stored into the appropriate low-order part of the word at the memory locations addressed by EA that cross a natural word boundary.

No Address Error exception occurs due to misalignment.

The SWL and SWR instructions are used in combination to store a word into memory locations that are not on a natural word boundary.

Exceptions

Address Error exception

Example

Assume register r9 contains 0x123_4567.

r9 | 0x0123 4567

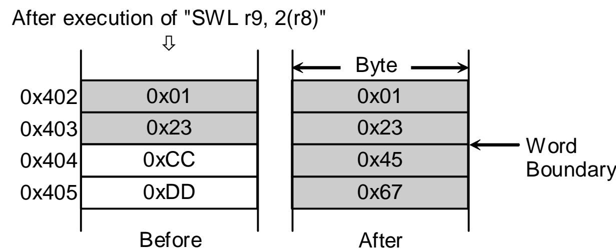
The following shows how to store the right portion of a general-purpose register after storing the left portion as described on the previous SWL pages.

- Big-endian mode

  The instruction:

  ```
  SWR r9,5(r8)
  ```

  starts at the rightmost byte in register r9 and stores that byte at address 0x0405. Then it stores bytes in register r9, going in the lower-address direction, until it reaches a word boundary in memory. The operation of this SWR instruction is as follows.



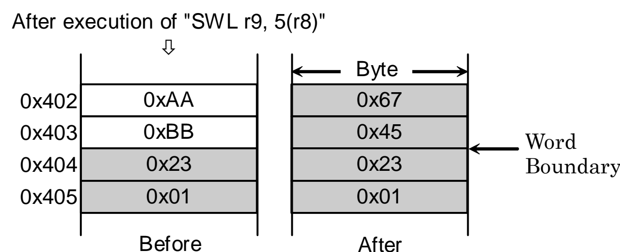
(a) Big-Endian

- Little-endian mode

  The instruction:

  ```
  SWR r9,2(r8)
  ```

  starts at the rightmost byte in register r9 and stores that byte at address 0x0402. Then it stores bytes in register r9, going in the higher-address direction, until it reaches a word boundary in memory. The operation of this SWR instruction is as follows.



(b) Little-Endian

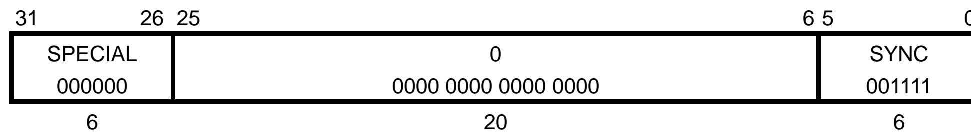
SYNC

Synchronize

Operation

Synchronize operation

Instruction Encoding



Description

The SYNC instruction interlocks the instruction pipeline until loads and stores performed prior to the present instruction are completed before any instructions after this instruction are allowed to start. See Section 5.2.4, *SYNC Instruction (32-Bit ISA)*.
If there is no data dependency, the TX19A continues to execute subsequent instructions. This is called *non-blocking loads*. By virtue of non-blocking loads, the instruction pipeline can continue to work on non-dependent instructions.

Exceptions

None

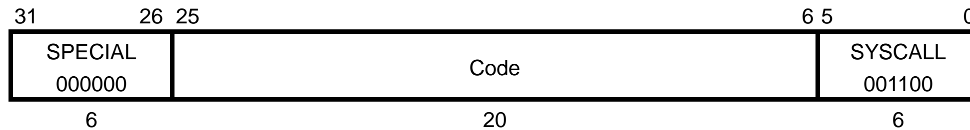
SYSCALL *code*

System Call

Operation

System call exception

Instruction Encoding



Description

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

The *code* field in a SYSCALL instruction is available for use as software parameters to pass additional information. To examine these bits, load the contents of the instruction at which the EPC register points. For details on System Call exceptions, see Section 9.1.10, *System Call Exception*.

Exceptions

System call exception

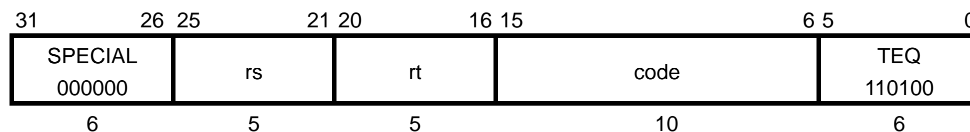
TEQ *rs*, *rt*, *code*

Trap If Equal

Operation

if *rs* = *rt* then Trap Exception; else Next Instruction

Instruction Encoding



Description

The contents of general-purpose register *rs* is compared to the contents of general-purpose register *rt*. If *rs* is equal to *rt*, a Trap exception occurs. The *code* field in a TEQ instruction is available for use as software parameters to pass additional information. To examine these bits, system software must load the instruction word from memory.

Exceptions

Trap exception

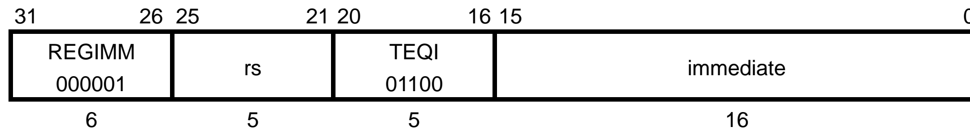
TEQI *rs*, *immediate*

Trap If Equal Immediate

Operation

if $rs = (immediate_{15})^{16} \parallel immediate_{15..0}$ then Trap Exception else Next Instruction

Instruction Encoding



Description

The 16-bit *immediate* is sign-extended and compared to the contents of general-purpose register *rs*. If *rs* is equal to *immediate*, a Trap exception occurs.

Exceptions

Trap exception

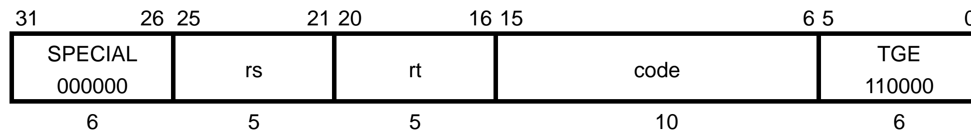
TGE *rs*, *rt*, *code*

Trap If Greater Than or Equal

Operation

if *rs* ≧ *rt* then Trap Exception; else Next Instruction

Instruction Encoding



Description

The contents of general-purpose register *rs* is compared to the contents of general-purpose register *rt* as signed integers. If *rs* is greater than or equal to *rt*, a Trap exception occurs. The *code* field in a TGE instruction is available for use as software parameters to pass additional information. To examine these bits, system software must load the instruction word from memory.

Exceptions

Trap exception

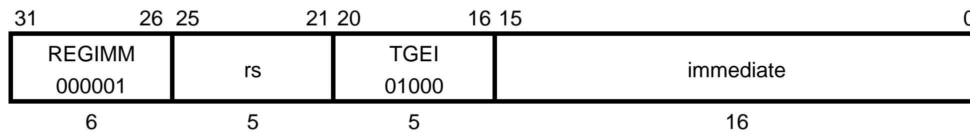
TGEI *rs*, *immediate*

Trap If Greater Than Or Equal Immediate

Operation

if $rs \geqq (immediate_{15})^{16} \| immediate_{15..0}$ then Trap Exception else Next Instruction

Instruction Encoding



Description

The 16-bit *immediate* is sign-extended and compared to the contents of general-purpose register *rs* as signed integers. If *rs* is greater than or equal to *immediate*, a Trap exception occurs.

Exceptions

Trap exception

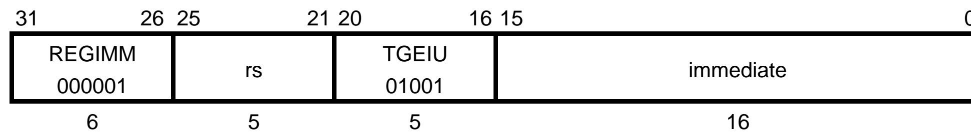
TGEIU *rs*, *immediate*

Trap If Greater Than Or Equal Immediate Unsigned

Operation

if $(0 \parallel rs) \geqq 0 \parallel (immediate_{15})^{16} \parallel immediate_{15..0}$ then Trap Exception else Next Instruction

Instruction Encoding



Description

The 16-bit *immediate* is sign-extended and compared to the contents of general-purpose register *rs* as unsigned integers. If *rs* is greater than or equal to *immediate*, a Trap exception occurs.

Exceptions

Trap exception

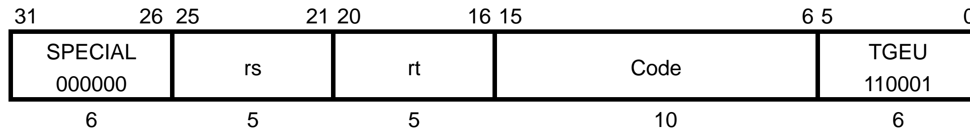
TGEU *rs*, *rt*, *code*

Trap If Greater Than or Equal Unsigned

Operation

if (0 || *rs*) ≧ (0 || *rt*) then Trap Exception; else Next Instruction

Instruction Encoding



Description

The contents of general-purpose register *rs* is compared to the contents of general-purpose register *rt* as unsigned integers. If *rs* is greater than or equal to *rt*, a Trap exception occurs. The *code* field in a TGEU instruction is available for use as software parameters to pass additional information. To examine these bits, system software must load the instruction word from memory.

Exceptions

Trap exception

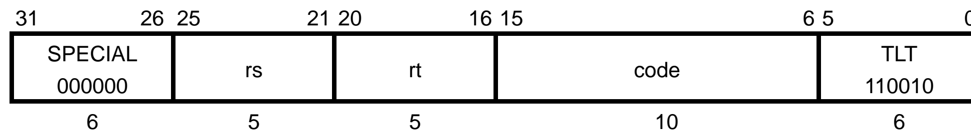
TLT *rs*, *rt*, *code*

Trap If Less Than

Operation

if *rs* < *rt* then Trap Exception; else Next Instruction

Instruction Encoding



Description

The contents of general-purpose register *rs* is compared to the contents of general-purpose register *rt* as signed integers. If *rs* is less than *rt*, a Trap exception occurs. The *code* field in a TLT instruction is available for use as software parameters to pass additional information. To examine these bits, system software must load the instruction word from memory.

Exceptions

Trap exception

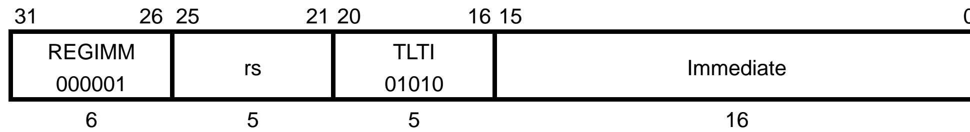
TLTI *rs*, *immediate*

Trap If Less Than Immediate

Operation

if $rs < (immediate_{15})^{16} \parallel immediate_{15..0}$ then Trap Exception else Next Instruction

Instruction Encoding



Description

The 16-bit *immediate* is sign-extended and compared to the contents of general-purpose register *rs* as signed integers. If *rs* is less than *immediate*, a Trap exception occurs.

Exceptions

Trap exception

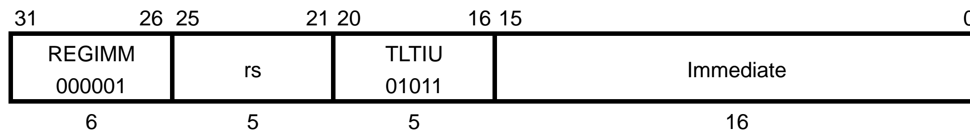
TLTIU *rs*, *immediate*

Trap If Less Than Immediate Unsigned

Operation

if $(0 \parallel rs) < 0 \parallel (immediate_{15})^{16} \parallel immediate_{15..0}$ then Trap Exception else Next Instruction

Instruction Encoding



Description

The 16-bit *immediate* is sign-extended and compared to the contents of general-purpose register *rs* as unsigned integers. If *rs* is less than *immediate*, a Trap exception occurs.

Exceptions

Trap exception

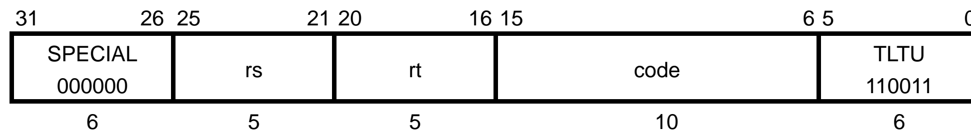
TLTU *rs*, *rt*, *code*

Trap If Less Than Unsigned

Operation

if (0 || *rs*) < (0 || *rt*) then Trap Exception; else Next Instruction

Instruction Encoding



Description

The contents of general-purpose register *rs* is compared to the contents of general-purpose register *rt* unsigned integers. If *rs* is less than *rt*, a Trap exception occurs. The *code* field in a TLTU instruction is available for use as software parameters to pass additional information. To examine these bits, system software must load the instruction word from memory.

Exceptions

Trap exception

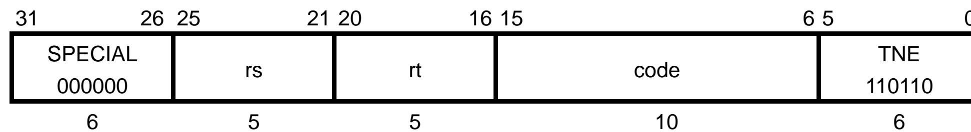
TNE *rs*, *rt*, *code*

Trap If Not Equal

Operation

if *rs* ≠ *rt* then Trap Exception; else Next Instruction

Instruction Encoding



Description

The contents of general-purpose register *rs* is compared to the contents of general-purpose register *rt*. If *rs* is not equal to *rt*, a Trap exception occurs. The *code* field in a TNE instruction is available for use as software parameters to pass additional information. To examine these bits, system software must load the instruction word from memory.

Exceptions

Trap exception

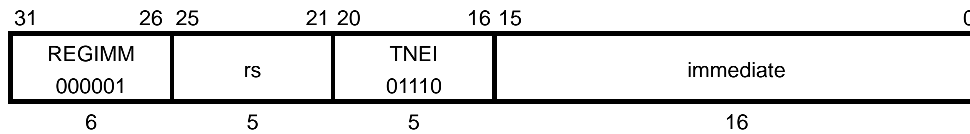
TNEI *rs*, *immediate*

Trap If Not Equal Immediate

Operation

if $rs \neq (immediate_{15})^{16} \parallel immediate_{15..0}$ then Trap Exception else Next Instruction

Instruction Encoding



Description

The 16-bit *immediate* is sign-extended and compared to the contents of general-purpose register *rs*. If *rs* is not equal to *immediate*, a Trap exception occurs.

Exceptions

Trap exception

WAIT

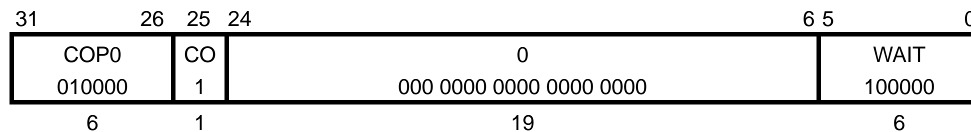
Enter Standby Mode

Operation

if Status[RP] = 1 then DOZE mode

else HALT mode

Instruction Encoding



Description

The WAIT instruction is used to stall the instruction pipeline to reduce the processor's power consumption. If the RP bit in the Status register is set, the processor enters DOZE mode. If the RP bit is cleared, the processor enters HALT mode. Refer to Chapter 10, *Low-Power Modes*.

The WAIT instruction must not be set in a delay slot of the branch or jump instruction.
Once the MTC0 instruction writes to the Status, EPC or ErrorEPC register, at least two instructions must be executed before the WAIT instruction. Otherwise, the operation is undefined.

Exceptions

Coprocessor Unusable exception

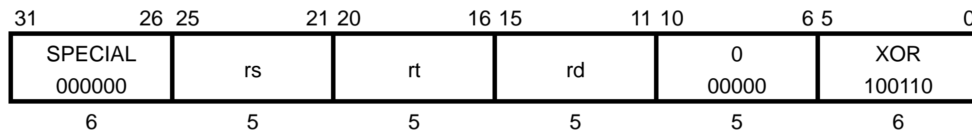
XOR *rd*, *rs*, *rt*

Exclusive OR

Operation

$rd \Leftarrow rs \text{ XOR } rt$

Instruction Encoding



Description

The contents of general-purpose register *rs* is exclusive-ORed with the contents of general-purpose register *rt*. The result is placed back into general-purpose register *rd*.

Exceptions

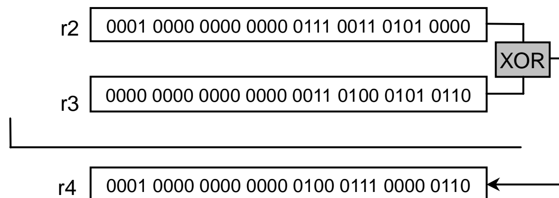
None

Example

Assume that registers r2 and r3 contain 0x1000_7350 and 0x0000_3456 respectively. Then, executing the instruction:

```
XOR r4,r2,r3
```

places 0x1000_4706 back in register r4, as shown below.



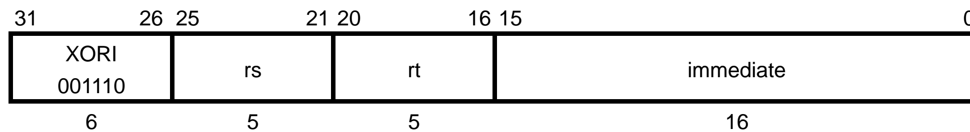
XORI *rt*, *rs*, *immediate*

Exclusive OR Immediate

Operation

$rt \Leftarrow rs \text{ XOR } (0^{16} \,\|\, immediate_{15..0})$

Instruction Encoding



Description

The 16-bit *immediate* is zero-extended and exclusive-ORed with the contents of general-purpose register *rs*. The result is placed back into *rt*.

The *immediate* field is 16 bits in length. If the *immediate* size is larger than that, you need to put it in a general-purpose register and use the XOR instruction (see Section 3.3.2, *32-Bit Constants*).

Exceptions

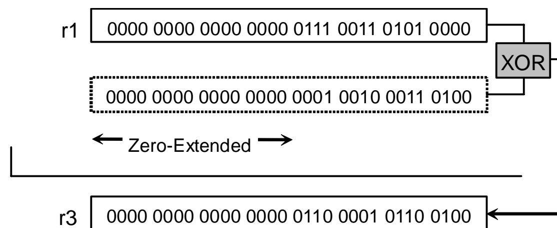
None

Example

Assume that register r2 contains 0x0000_7350. Then, executing the instruction:

```
XORI r3,r2,0x1234
```

places 0x0000_6164 back in register r3, as shown below.



Appendix B 16-Bit ISA Details

This appendix presents detailed information concerning each instruction in the 16-bit ISA. Each instruction is listed alphabetically by mnemonic. Each listing contains complete information about assembler syntax, instruction format, operation and exceptions that may occur due to the execution of the instruction. For the variations of instruction formats, see Section 4.1, *Instruction Formats*.

To fit within the 16-bit limit, the register fields (*rx*, *ry*, *rz* and *base*) in the 16-bit instructions are only 3 bits. Therefore, to the 16-bit instructions, only eight of the 32 general-purpose registers are normally visible, r2 to r7, r16 and r17. These registers are encoded as follows.

Code	Register	Code	Register
000	r16	100	r4
001	r17	101	r5
010	r2	110	r6
011	r3	111	r7

Additionally, specific instructions implicitly reference r24 (t8), r28 (gp), r29 (sp), r30 (fp) and r31 (ra). r24 serves as the condition code register for handling compare results. r28 is the global pointer. r29 maintains the program stack pointer. r30 is the frame pointer. r31 is the link register to store the subroutine return address. These registers are implicitly referred to through special function codes.

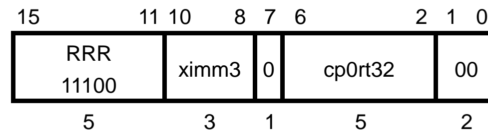
AC0IU *cp0rt32*, *imm3*

Add Coprocessor 0 Immediate Unsigned

Operation

cp0rt32 ⇐ *cp0rt32* + *imm3*

Instruction Encoding



Description

The encoding used for the 3-bit *imm3* field is shown below. The value indicated by *imm3* is added to the contents of CP0 register *cp0rt32*. The result is placed back into *cp0rt32*. *imm3* can only be one of these: –8, –4, –2, –1, +1, +2, +4, +8.

ximm3	imm3
1 1 1	−8
1 1 0	−4
1 0 1	−2
1 0 0	−1
0 0 0	+1
0 0 1	+2
0 1 0	+4
0 1 1	+8

No Integer Overflow exception occurs under any circumstances.

Once the AC0IU instruction writes to the Status, EPC or ErrorEPC register, at least two instructions must be executed before the ERET instruction. Otherwise, the operation is undefined.

The AC0IU instruction that modifies the contents of the SSCR register must be followed by two NOPs.

Exceptions

Coprocessor Unusable exception

Example

Assume that the EPC register contains 0x8001_0060. Then, the instruction:

```
AC0IU  EPC, -8
```

writes the result of 0x8001_0058 into EPC.

ADDIU fp, *immediate*

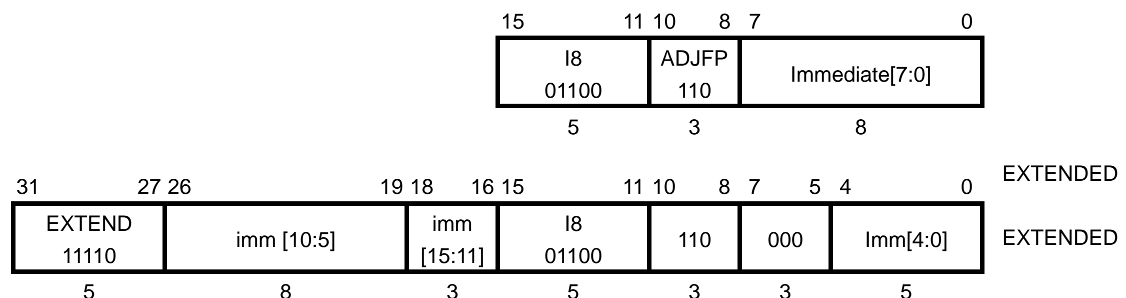
Add Immediate Unsigned

Operation

$r30 \Leftarrow r30 + (immediate_{7})^{22} \,||\, (immediate_{7..0}) \,||\, 00$

(EXTENDED) $r30 \Leftarrow r30 + (immediate_{15})^{16} \,||\, (immediate_{15..0})$

Instruction Encoding



Description

The term "unsigned" in the instruction name is a misnomer. The 8-bit *immediate* is shifted left by two bits and *sign-extended*. The resultant value is added to the contents of the fp (r30) register.

No Integer Overflow exception occurs under any circumstances.

Since the 8-bit *immediate* is shifted left by two bits, the immediate range is –512 to +504, in increments of four. If the *immediate* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of –32768 to +32767. When EXTENDed, the *immediate* operand is not shifted at all.

Exceptions

None

Example

Assume that frame pointer register fp contains 0x0000_2000. Then, the instruction:

```
ADDIU fp,8
```

places the result 0x0000_2008 in fp.

ADDIU *rx*, *immediate*

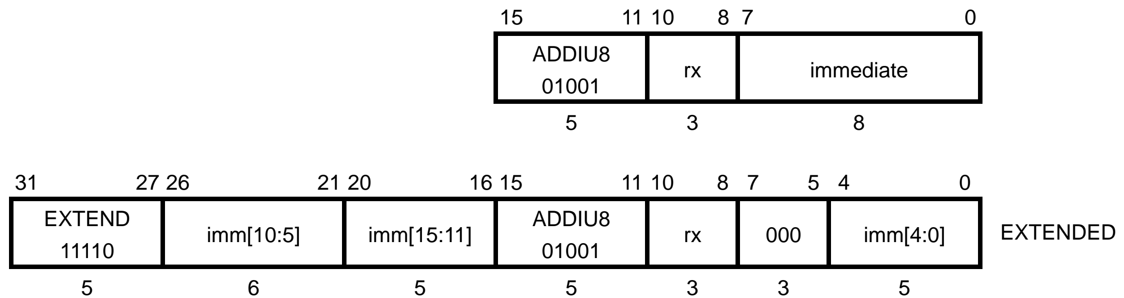
Add Immediate Unsigned

Operation

$rx \Leftarrow rx + (immediate_7)^{24} \parallel (immediate_{7..0})$

(EXTENDED) $rx \Leftarrow rx + (immediate_{15})^{16} \parallel (immediate_{15..0})$

Instruction Encoding



Description

The term "unsigned" in the instruction name is a misnomer. The 8-bit *immediate* is *sign-extended* and added to the contents of general-purpose register *rx*. The result is placed back into general-purpose register *rx*.

No Integer Overflow exception occurs under any circumstances.

With the 8-bit *immediate* field, the immediate range is -128 to +127. If the *immediate* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

None

ADDIU *rx*, pc, *immediate*

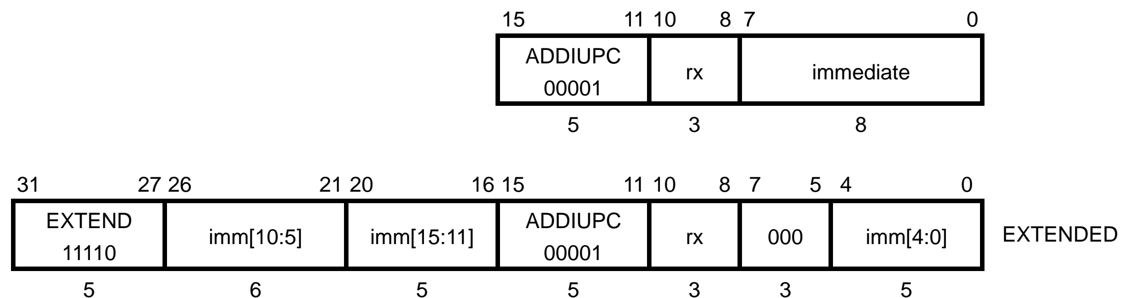
Add Immediate Unsigned

Operation

$rx \Leftarrow$ Masked base PC + 0^{22} || ($immediate_{7..0}$) || 00

(EXTENDED) $rx \Leftarrow$ Masked base PC + $(immediate_{15})^{16}$ || ($immediate_{15..0}$)

Instruction Encoding



Description

The PC value used as the base for address calculation is called base PC value. The two low-order bits of the PC are cleared to form a "masked base PC value." The 8-bit *immediate* is shifted left by two bits, *zero*-extended and then added to the masked base PC value to form a virtual address. This address is placed into general-purpose register *rx*. This instruction is used to calculate the PC relative address of an instruction or data in its proximity and place it in a register.

No Integer Overflow exception occurs under any circumstances.

The 32-bit PC-relative instruction is not a valid 32-bit ISA instruction; thus the operation of this instruction differs from that of the ADDIU instruction in the 32-bit ISA.

Since the 8-bit *immediate* is shifted left by two bits, the immediate range is 0 to 1020, in increments of four. If the *immediate* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767. When EXTENDed, the *immediate* operand is not shifted at all.

The base PC value differs as follows, depending on whether this instruction is in a delay slot and whether it is prepended with an EXTEND prefix.

ADDIUPC	Base PC Value
Delay slot of a JR or JALR instruction	Address of the JR or JALR instruction
Delay slot of a JAL or JALX instruction	Address of the upper halfword of the JAL or JALX instruction
EXTENDed	Address of the EXTEND instruction code
Not EXTENDed	Address of the ADDIUPC instruction

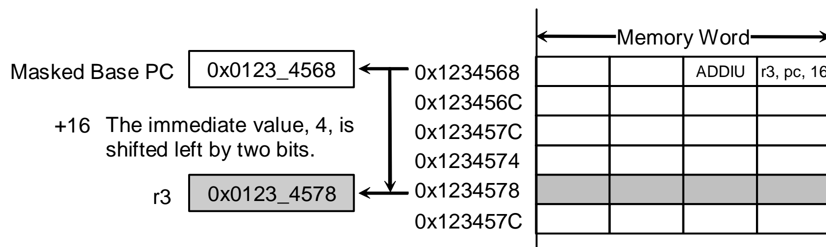
Exceptions

None

Example

```
ADDIU r3,pc,16
```

Assume that this instruction is at address 0x0123_456A which is not a delay slot. Then, the masked PC value of 0x0123_4568 is obtained by clearing its two low-order bits. Since the immediate value is shifted left by two bits by the processor hardware, the assembler turns the specified operand (16) into a code of 4. Thus the instruction code for this ADDIU instruction becomes 0x0B04. The offset is added to the masked PC value as shown below, and the result is placed in register r3.



ADDIU *rx*, sp, *immediate*

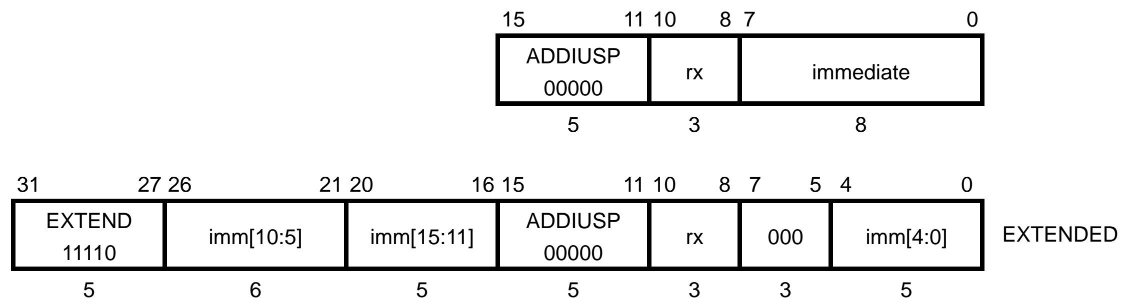
Add Immediate Unsigned

Operation

$rx \Leftarrow \text{sp} + 0^{22} \parallel (immediate_{7..0}) \parallel 00$

(EXTENDED) $rx \Leftarrow \text{sp} + (immediate_{15})^{16} \parallel (immediate_{15..0})$

Instruction Encoding



Description

In this instruction format, the 8-bit *immediate* is shifted left by two bits and *zero-extended*. The resultant value is added to the contents of stack pointer register sp (r29), and the result is placed into general-purpose register *rx*.

No Integer Overflow exception occurs under any circumstances.

Since the 8-bit *immediate* is shifted left by two bits, the immediate range is 0 to 1020, in increments of four. If the *immediate* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767. When EXTENDed, the *immediate* operand is not shifted at all.

Exceptions

None

ADDIU *ry*, *rx*, *immediate*

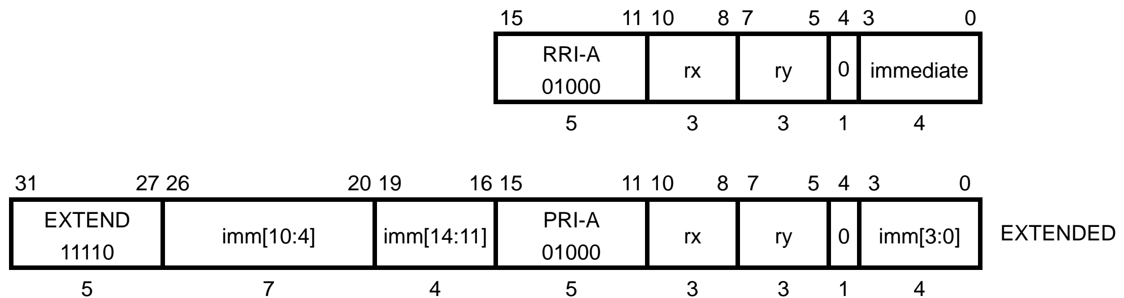
Add Immediate Unsigned

Operation

$ry \Leftarrow rx + (immediate_{3})^{28} \parallel (immediate_{3..0})$

(EXTENDED) $ry \Leftarrow rx + (immediate_{14})^{17} \parallel (immediate_{14..0})$

Instruction Encoding



Description

The term "unsigned" in the instruction name is a misnomer. The 4-bit immediate is *sign-extended* and added to the contents of general-purpose register *rx*. The result is placed into general-purpose register *ry*.

No Integer Overflow exception occurs under any circumstances.

With the 4-bit *immediate* field, the immediate range is -8 to +7. If the immediate is outside this range, the instruction is EXTENDed to provide a 15-bit signed immediate in the range of -16384 to +16833.

Exceptions

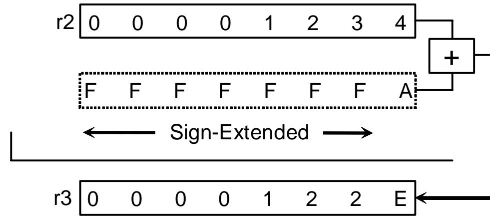
None

Example

Assume that register r2 contains 0x0000_1234. Then, executing the instruction:

```
ADDIU r3,r2,-6
```

places the sum 0x0000_122E into r3.



ADDIU sp, *immediate*

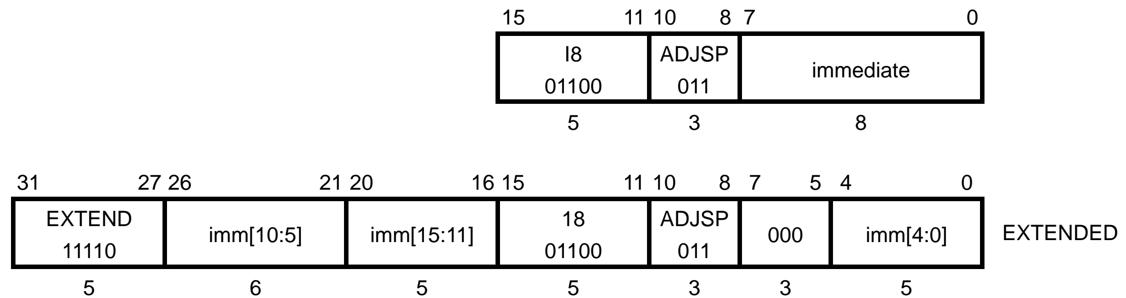
Add Immediate Unsigned

Operation

$sp \Leftarrow sp + (immediate_{7})^{21} \parallel (immediate_{7..0}) \parallel 000$

(EXTENDED) $sp \Leftarrow sp + (immediate_{15})^{16} \parallel (immediate_{15..0})$

Instruction Encoding



Description

The term "unsigned" in the instruction name is a misnomer. The 8-bit *immediate* is shifted left by three bits and *sign-extended*. The resultant value is added to the contents of stack pointer register sp (r29).

No Integer Overflow exception occurs under any circumstances.

Since the 8-bit *immediate* is shifted left by three bits, the immediate range is -1024 to +1016, in increments of eight. If the *immediate* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767. When EXTENDed, the *immediate* operand is not shifted at all.

Exceptions

None

Example

Assume stack pointer register sp contains 0x0000_2000. Then, the instruction:

```
ADDIU sp,8
```

places the result 0x0000_2008 in sp, as shown below.

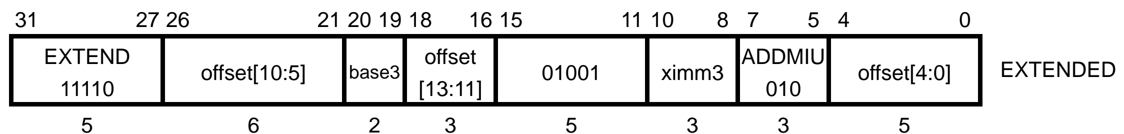
ADDMIU *offset* (*base3*), *imm3*

Add Immediate to Memory Word

Operation

{zero-extend (*offset* || 00) + (*base3*)} ⇐ {zero-extend (*offset* || 00) + (*base3*)} + *imm3*

Instruction Encoding



Description

The 14-bit *offset* is shifted left by two bits, zero-extended, then added to the contents of the general purpose register specified by *base3* to form an effective address (EA). The value indicated by the 3-bit *imm3* is added to the memory word addressed by the EA, and the sum is written back to the EA.

base3	GPR
01	r28 (gp)
10	r29 (sp)
11	r30 (fp)

imm3 can only be one of these: –8, –4, –2, –1, +1, +2, +4, +8.

ximm3	imm3
1 1 1	−8
1 1 0	−4
1 0 1	−2
1 0 0	−1
0 0 0	+1
0 0 1	+2
0 1 0	+4
0 1 1	+8

No Integer Overflow exception occurs under any circumstances.

Since the 14-bit *offset* is shifted left by two bits, the offset range is 0 to 65532, in increments of four.

Exceptions

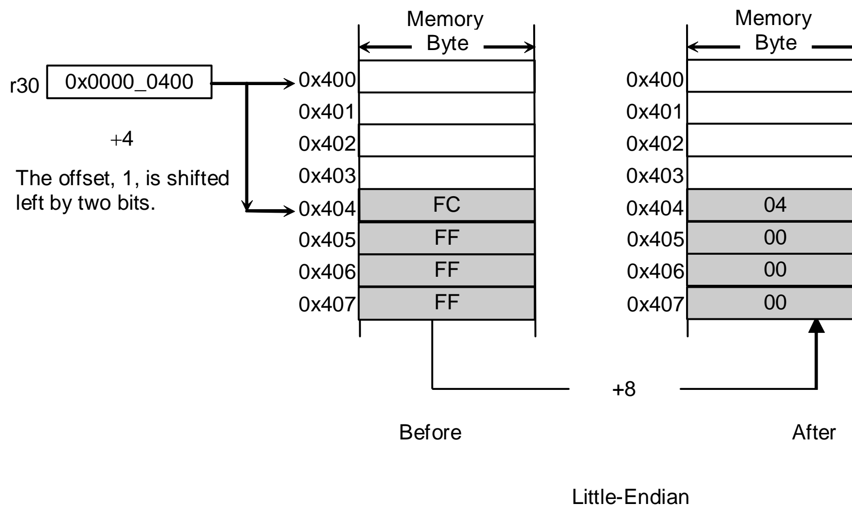
Address Error exception

Example

Assume that the fp register contains 0x0000_0400 and that the memory word at address 0x0404 is 0xFFFF_FFFC. Then, the instruction:

```
ADDMIU 4(fp), 8
```

adds 8 to the contents of the memory word at 0x404, as shown below.



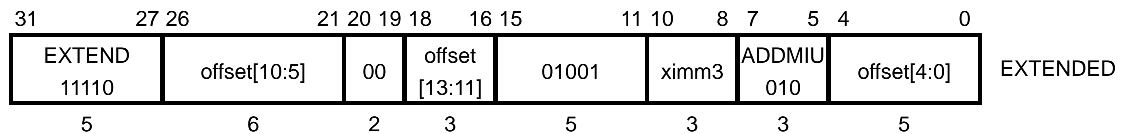
ADDMIU *offset* (r0), *imm3*

Add Immediate to Memory Word

Operation

$\{\text{sign-extend}\ (\mathit{offset} \parallel 00)\} \Leftarrow \{\text{sign-extend}\ (\mathit{offset} \parallel 00)\} + \mathit{imm3}$

Instruction Encoding



Description

The 14-bit *offset* is shifted left by two bits, sign-extended, then added to the contents of general purpose register r0 to form an effective address (EA). The value indicated by the 3-bit *imm3* is added to the memory word addressed by the EA, and the sum is written back to the EA.

imm3 can only be one of these: –8, –4, –2, –1, +1, +2, +4, +8.

ximm3	imm3
1 1 1	−8
1 1 0	−4
1 0 1	−2
1 0 0	−1
0 0 0	+1
0 0 1	+2
0 1 0	+4
0 1 1	+8

No Integer Overflow exception occurs under any circumstances.

Since the 14-bit *offset* is shifted left by two bits, the offset range is –32768 to +32764, in increments of four.

Exceptions

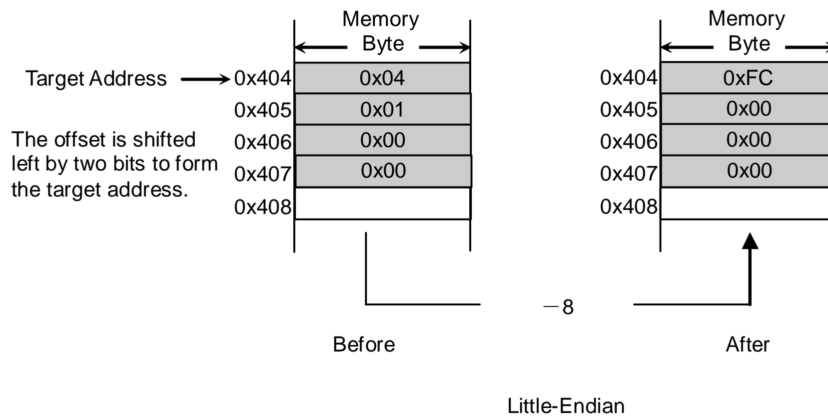
Address Error exception

Example

Assume that the memory word at address 0x0404 is 0x0000_0104. Then, the instruction:

```
ADDMIU 0x404(r0), -8
```

adds –8 to the contents of the memory word at 0x404, as shown below:



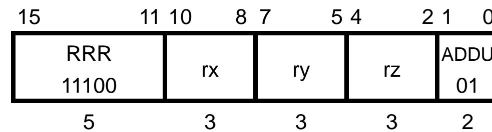
ADDU *rz*, *rx*, *ry*

Add Unsigned

Operation

$rz \Leftarrow rx + ry$

Instruction Encoding



Description

The contents of general-purpose register *rx* is added to the contents of general-purpose register *ry*, and the result is placed into general-purpose register *rz*. No Integer Overflow exception occurs under any circumstances.

Exceptions

None

Example

Assume that registers r2 and r3 contain 0x2000_0000 and 0x0123_4567 respectively. Then, executing the instruction:

```
ADDU r4,r2,r3
```

places the sum (0x323_4567) into r4.

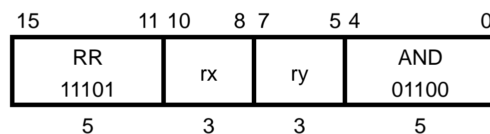
AND *rx*, *ry*

AND

Operation

$rx \Leftarrow rx$ AND ry

Instruction Encoding



Description

The contents of general-purpose register *rx* is ANDed with the contents of general-purpose register *ry*, and the result is placed back into general-purpose register *rx*.

Exceptions

None

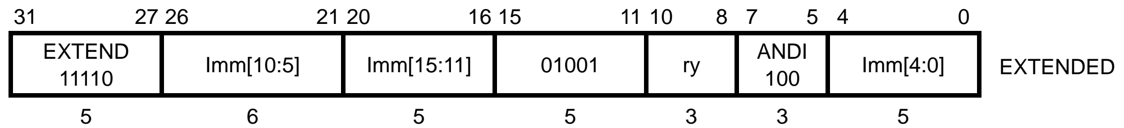
ANDI *ry*, *immediate*

Logical AND Immediate

Operation

$ry \Leftarrow ry$ AND $(0^{16} \parallel immediate_{15..0})$

Instruction Encoding



Description

The 16-bit *immediate* is zero-extended and ANDed with the contents of general-purpose register *ry*. The result is placed back into *ry*.

The *immediate* field is 16 bits in length. If the immediate size is larger than that, you need to put it in a general-purpose register and use the AND instruction (see 3.3.2, *32-Bit Constants*).

Exceptions

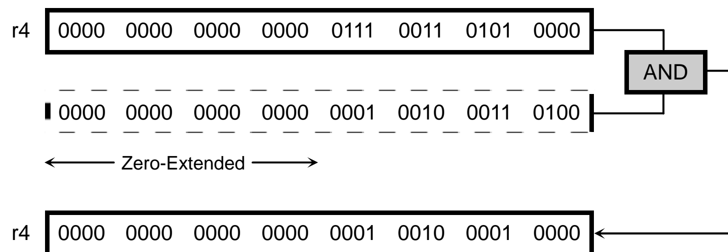
None

Example

Assume that register r4 contains 0x0000_7350. Then, the instruction:

```
ANDI r4,0x1234
```

performs the logical AND between 0x0000_7350 and 0x0000_1234 and puts the result (0x0000_1210) in r4, as shown below.



B *offset*

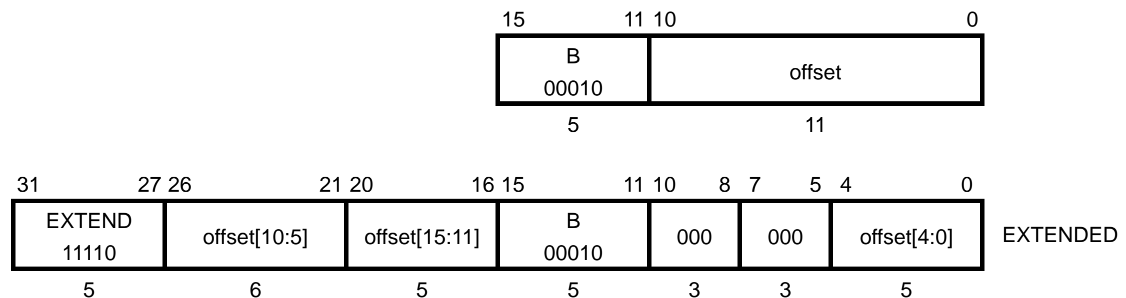
Unconditional Branch

Operation

$pc \Leftarrow pc + 2 + \text{sign-extend}(\mathit{offset} \parallel 0)$

(EXTENDED) $pc \Leftarrow pc + 4 + \text{sign-extend}(\mathit{offset} \parallel 0)$

Instruction Encoding



Description

The program unconditionally branches to the target address with a delay of one instruction (or two pipeline cycles). See Section 5.3.4, *Branch Instructions (16-Bit ISA)*, for pipeline delays. This instruction does not have a delay slot. If the branch is taken, the instruction that immediately follows this instruction is not executed. The target address is computed relative to the address of the immediately following instruction, i.e., PC+2 when the instruction is not EXTENDED and PC+4 when EXTENDed.

Since the 11-bit *offset* is shift left by one bit, the branch range is -2048 to +2046. If the *offset* is outside this range, the instruction is EXTENDed to provide a 17-bit signed immediate in the range of -65536 to +65534. Whether EXTENDed or not, the target address is computed in the same manner.

Exceptions

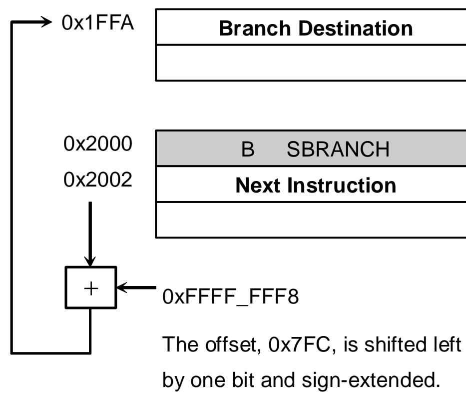
None

Example

```
B SBRANCH
```

Assume that this branch instruction resides at address 0x2000 and that label SBRANCH points to absolute address 0x1FFA. Then the assembler/linker turns this label into an offset operand of 0x7FC (see the figure below). Thus the instruction code for this branch instruction becomes 0x17FC.

The processor unconditionally transfers program control to address 0x1FFA. The instruction following the B instruction is never executed.



BAL *offset*

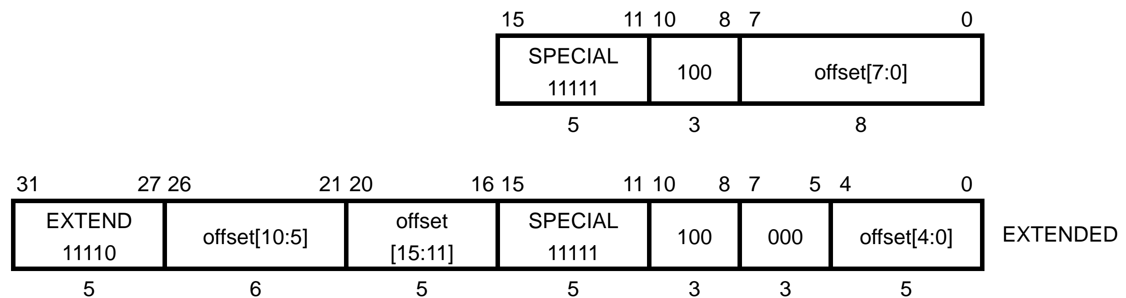
Branch And Link

Operation

r31 ⇐ pc + 3; pc ⇐ pc + 2 + sign-extend (*offset* || 0)

(EXTENDED) r31 ⇐ pc + 5; pc ⇐ pc + 4 + sign-extend (*offset* || 0)

Instruction Encoding



Description

The program unconditionally branches to the target address with a delay of one instruction (or two pipeline cycles). See Section 5.3.4, *Branch Instructions (16-Bit ISA)*, for pipeline delays. This instruction does not have a delay slot. If the branch is taken, the instruction that immediately follows this instruction is not executed. The target address is computed relative to the address of the immediately following instruction, i.e., PC+2 when the instruction is not EXTENDED and PC+4 when EXTENDed.

The address of the instruction following the BAL instruction (PC+2) is saved in the link register, r31 (ra). The least-significant bit of r31 stores the ISA mode bit that was in effect before the branch (16-bit ISA = 1).

Since the 8-bit *offset* is shifted left by one bit, the branch range is –256 to +254. If the *offset* is outside this range, the instruction is EXTENDed to provide a 17-bit signed immediate in the range of –65536 to –65534. In this case also, the target address is computed the same way.

Exceptions

None

Example

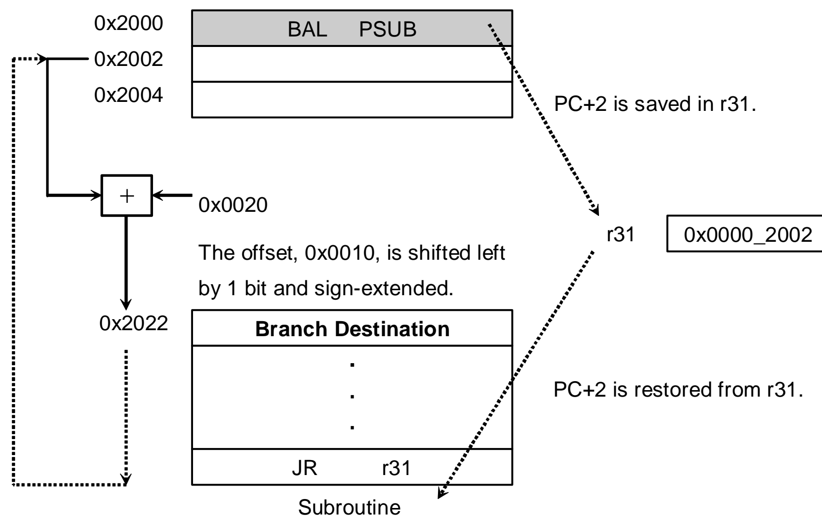
```
BAL PSUB
```

Assume that this branch instruction resides at address 0x2000 and that label PSUB points to absolute address 0x2022. Then, the assembler/linker turns this label into an offset operand of 0x0010 (see the figure below).

The program unconditionally branches to address 0x2022.

The JR instruction is used at the end of the called subroutine to return control to the instruction after the BAL instruction.

```
JR r31
```



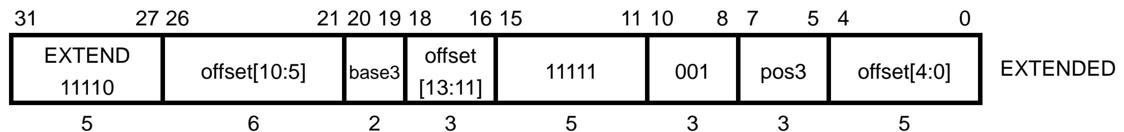
BCLR *offset* (*base3*), *pos3*

Bit Clear

Operation

{zero-extend (*offset*) + (*base3*)} [*pos3*] ⇐ 0

Instruction Encoding



Description

A bit specified by *pos3* in a memory byte is cleared. The effective address is computed by zero-extending the 14-bit *offset* and adding the resultant value to the contents of a general-purpose register indicated by *base3*. The encoding used for *base3* is as follows:

base3	GPR
01	gp(r28)
10	sp(r29)
11	fp(r30)

With the 14-bit *offset* field, the offset range is 0 to +16383.

Exceptions

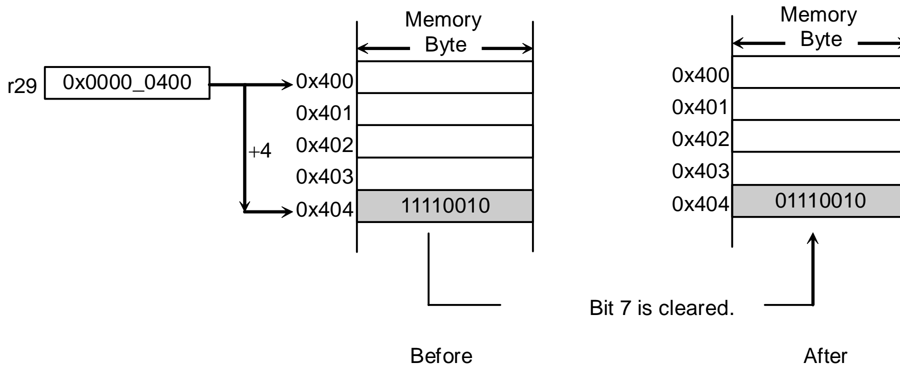
Address Error exception

Example

Assume that the sp register (r29) contains 0x0000_0400 and that the byte position at address 0x0404 contains 0xF2. Then, the instruction:

```
BCLR 4(sp),7
```

clears bit 7 of byte data 0xF2 as shown below.



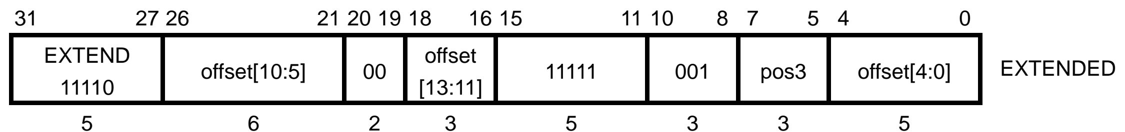
BCLR *offset* (r0), *pos3*

Bit Clear

Operation

{sign-extend (*offset*)} [*pos3*] $\Leftarrow$ 0

Instruction Encoding



Description

A bit specified by *pos3* in a memory byte is cleared. The effective address is computed by sign-extending the 14-bit *offset* and adding the resultant value to the contents of general-purpose register r0, which is hardwired to a value of zero.
With the 14-bit *offset* field, the offset range is -8192 to +8191.

Exceptions

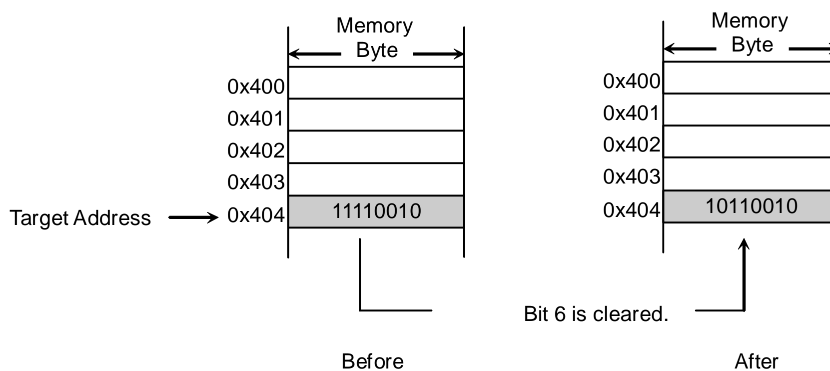
Address Error exception

Example

Assume that the byte position at address 0x0404 contains 0xF2. Then, the instruction:

```
BCLR 0x404(r0), 6
```

clears bit 6 of byte data 0xF2 as shown below.



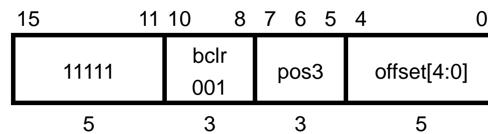
BCLR *offset* (fp), *pos3*

Bit Clear

Operation

{zero-extend (*offset*) + (fp)} [*pos3*] ⇐ 0

Instruction Encoding



Description

A bit specified by *pos3* in a memory byte is cleared. The effective address is computed by zero-extending the 5-bit *offset* and adding the resultant value to the contents of the fp register (r30). With the 5-bit *offset* field, the offset range is 0 to +31.

Exceptions

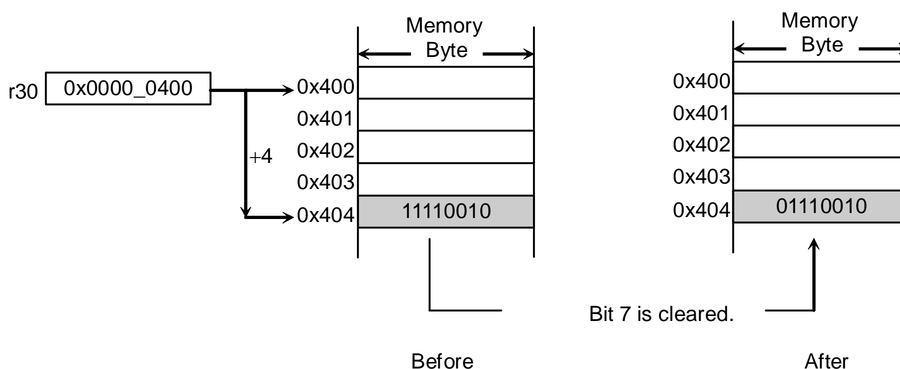
Address Error exception

Example

Assume that the fp register (r30) contains 0x0000_0400 and that the byte position at address 0x0404 contains 0xF2. Then, the instruction:

```
BCLR 4(fp), 7
```

clears bit 7 of byte data 0xF2 as shown below.



BEQZ *rx*, *offset*

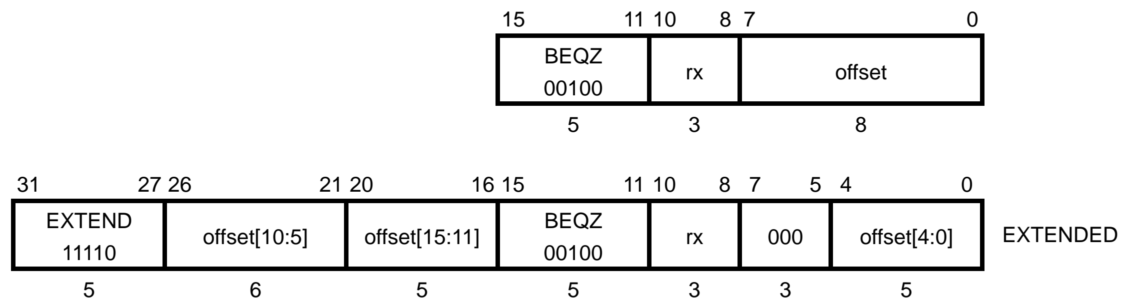
Branch On Equal To Zero

Operation

if *rx* = 0 then pc ⇐ pc + 2 + sign-extend (*offset* || 0)

(EXTENDED) if *rx* = 0 then pc ⇐ pc + 4 + sign-extend (*offset* || 0)

Instruction Encoding



Description

If the contents of general-purpose register *rx* is equal to zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). See Section 5.3.4, *Branch Instructions (16-Bit ISA)*, for pipeline delays. This instruction does not have a delay slot. If the branch is taken, the instruction that immediately follows this instruction is not executed. The target address is computed relative to the address of the immediately following instruction, i.e., PC+2 when the instruction is not EXTENDED and PC+4 when EXTENDed.

Since the 8-bit *offset* is shifted left by one bit, the branch range is -256 to +254. If the *offset* is outside this range, the instruction is EXTENDed to provide a 17-bit signed immediate in the range of -65536 to +65534. Whether EXTENDed or not, the target address is computed in the same manner.

Exceptions

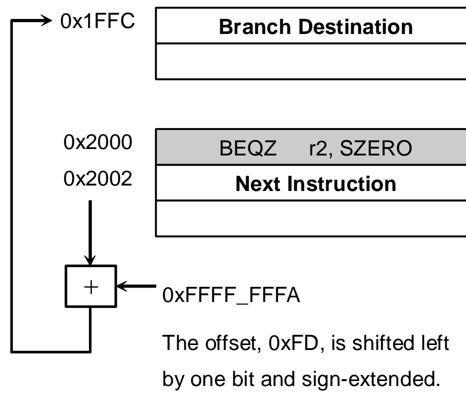
None

Example

```
BEQZ r2,SZERO
```

Assume that this branch instruction resides at address 0x2000 and that label SZERO points to absolute address 0x1FFC. Then the assembler/linker turns this label into an offset operand of 0xFD (see the figure below). Thus the instruction code for this branch instruction becomes 0x22FD.

If the contents of r2 are equal to zero, the processor transfers program control to address 0x1FFC. Otherwise, the program just continues to the next instruction at 0x2002.



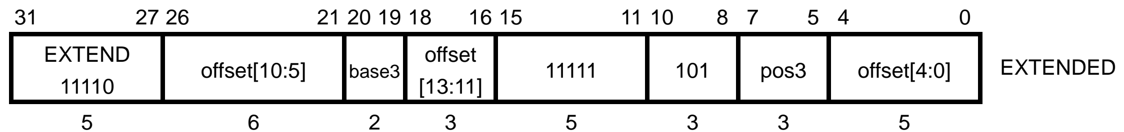
BEXT *offset* (*base3*), *pos3*

Bit Extract

Operation

t8 ⇐ 31'b 000_0000_0000_0000_0000_0000_0000_0000 || {zero-extend (*offset*) + (*base3*)} [*pos3*]

Instruction Encoding



Description

A bit specified by *pos3* in a memory byte is copied into the least-significant bit (LSB) of general purpose register t8 (r24). The upper 31 bits of t8 are filled with zeros. The effective address is computed by zero-extending the 14-bit *offset* and adding the resultant value to the contents of a general-purpose register indicated by *base3*. The encoding used for *base3* is as follows:

base3	GPR
01	gp(r28)
10	sp(r29)
11	fp(r30)

With the 14-bit *offset* field, the offset range is 0 to +16383.

Exceptions

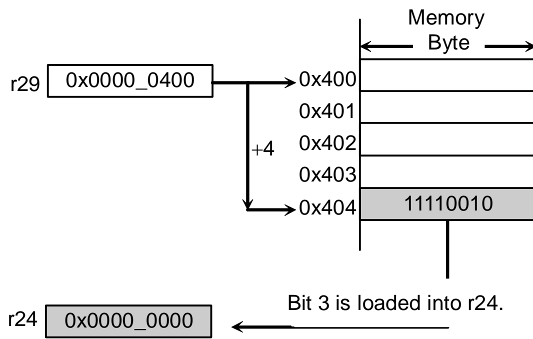
Address Error exception

Example

Assume that the sp register (r29) contains 0x0000_0400 and that the byte position at address 0x0404 contains 0xF2. Then, the instruction:

```
BEXT 4(sp), 3
```

loads r24 with 0x0000_0000.



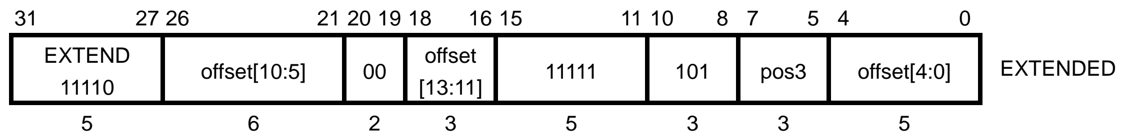
BEXT *offset* (r0), *pos3*

Bit Extract

Operation

t8 ⇐ 31'b 000_0000_0000_0000_0000_0000_0000_0000 || {sign-extend (*offset*)} [*pos3*]

Instruction Encoding



Description

A bit specified by *pos3* in a memory byte is copied into the least-significant bit (LSB) of general purpose register t8 (r24). The upper 31 bits of t8 are filled with zeros. The effective address is computed by sign-extending the 14-bit *offset* and adding the resultant value to the contents of general-purpose register r0, which is hardwired to a value of zero.

With the 14-bit *offset* field, the offset range is -8192 to +8191.

Exceptions

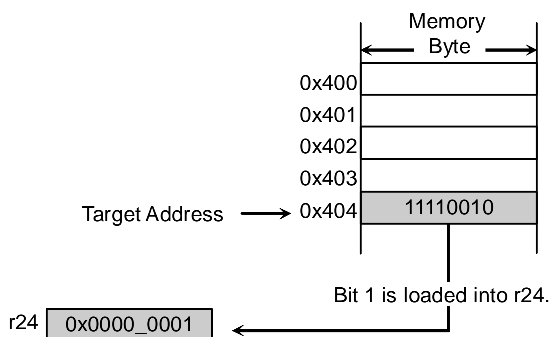
Address Error exception

Example

Assume that the byte position at address 0x0404 contains 0xF2. Then, the instruction:

```
BEXT 0x404(r0), 1
```

loads r24 with 0x0000_0001.



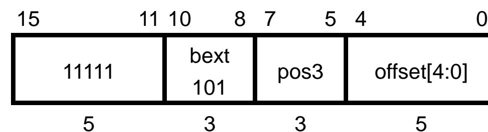
BEXT *offset* (fp), *pos3*

Bit Extract

Operation

t8 ⇐ 31'b 000_0000_0000_0000_0000_0000_0000_0000 || {zero-extend (*offset*) + (fp)} [*pos3*]

Instruction Encoding



Description

A bit specified by *pos3* in a memory byte is copied into the least-significant bit (LSB) of general purpose register t8 (r24). The upper 31 bits of t8 are filled with zeros. The effective address is computed by zero-extending the 5-bit *offset* and adding the resultant value to the contents of the fp register (r30).

With the 5-bit *offset5* field, the offset range is 0 to +31.

Exceptions

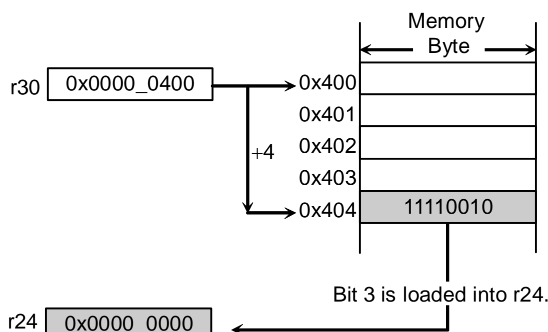
Address Error exception

Example

Assume that the fp register (r30) contains 0x0000_0400 and that the byte position at address 0x0404 contains 0xF2. Then, the instruction:

```
BEXT 4(fp), 3
```

loads r24 with 0x0000_0000.



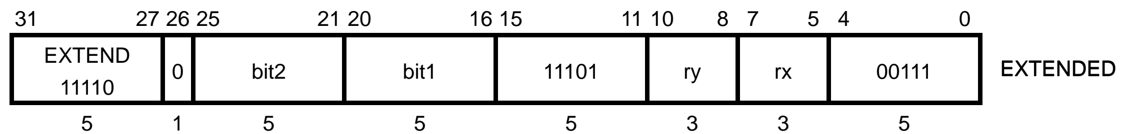
BFINS *ry*, *rx*, *bit2*, *bit1*

Bit Field Insert

Operation

$ry[bit2{:}bit1] \Leftarrow rx[bit2\text{-}bit1 : 0]$;

Instruction Encoding



Description

A bit field indicated by [(*bit2* – *bit1*):0] in general-purpose register *rx* is copied into a location indicated by (*bit2*:*bit1*) in general-purpose register *ry*.

Exceptions

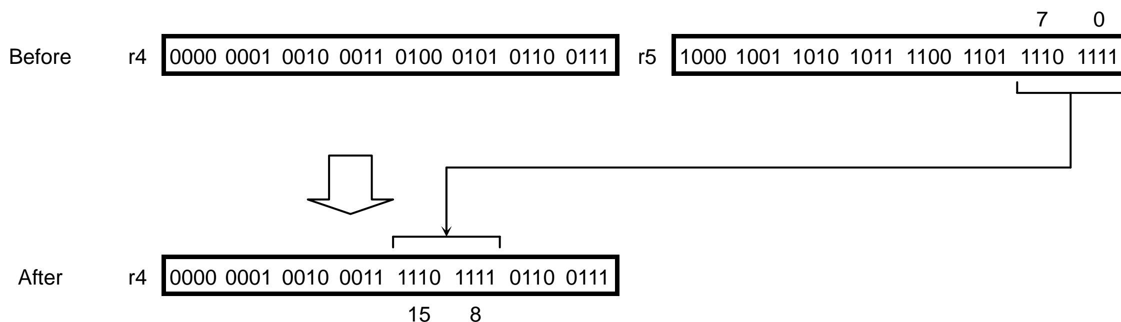
None

Example

Assume that general-purpose registers r4 and r5 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
bfins  r4, r5, 15, 8
```

reads bits 7-0 in r5 and deposits them in bits 15-8 in r4, as shown below.



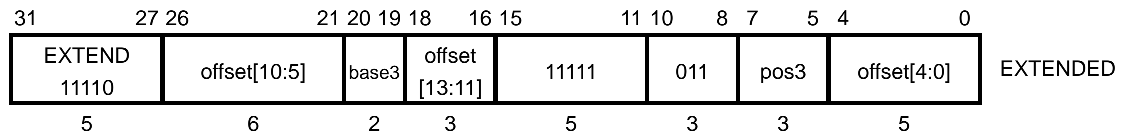
BINS *offset* (*base3*), *pos3*

Bit Insert

Operation

{zero-extend (*offset*) + (*base3*)} [*pos3*] ⇐ t8[0]

Instruction Encoding



Description

The least-significant bit (LSB) of general-purpose register t8 (r24) is copied into a bit position indicated by *pos3* in a memory byte. The effective address is computed by zero-extending the 14-bit *offset* and adding the resultant value to the contents of a general-purpose register indicated by *base3*. The encoding used for *base3* is as follows:

base3	GPR
01	gp(r28)
10	sp(r29)
11	fp(r30)

With the 14-bit *offset* field, the offset range is 0 to +16383.

Exceptions

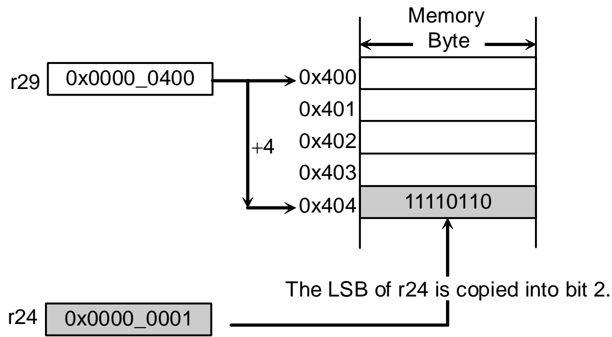
Address Error exception

Example

Assume that the sp register (r29) contains 0x0000_0400, that the byte position at address 0x0404 contains 0xF2 and that r24 contains 0x0000_0001. Then, the instruction:

```
BINS 4(sp), 2
```

replaces bit 2 at address 0x0404 with a 1.



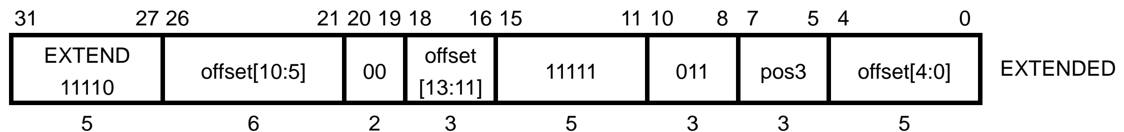
BINS *offset* (r0), *pos3*

Bit Insert

Operation

{sign-extend (*offset*)} [*pos3*] ⇐ t8[0]

Instruction Encoding



Description

The least-significant bit (LSB) of general-purpose register t8 (r24) is copied into a bit position indicated by *pos3* in a memory byte. The effective address is computed by sign-extending the 14-bit *offset* and adding the resultant value to the contents of general-purpose register r0, which is hardwired to a value of zero.

With the 14-bit *offset* field, the offset range is -8192 to +8191.

Exceptions

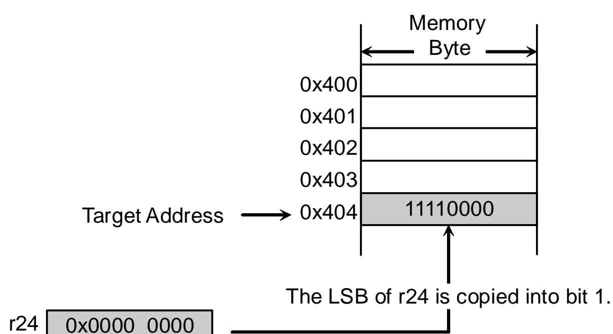
Address Error exception

Example

Assume that the byte position at address 0x0404 contains 0xF2 and that r24 contains 0x0000_0000. Then, the instruction:

```
BINS 0x404(r0), 1
```

replaces bit 1 at address 0x0404 with a 0.



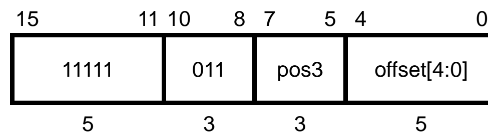
BINS *offset* (fp), *pos3*

Bit Insert

Operation

{zero-extend (*offset*) + (fp)} [*pos3*] ⇐ t8[0]

Instruction Encoding



Description

The least-significant bit (LSB) of general-purpose register t8 (r24) is copied into a bit position indicated by *pos3* in a memory byte. The effective address is computed by zero-extending the 5-bit *offset* and adding the resultant value to the contents of the fp register (r30).

With the 5-bit *offset* field, the offset range is 0 to +31.

Exceptions

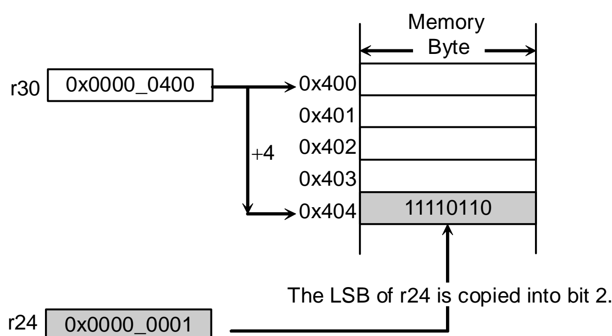
Address Error exception

Example

Assume that the fp register (r30) contains 0x0000_0400, that the byte position at address 0x0404 contains 0xF2 and that r24 contains 0x0000_0001. Then, the instruction:

```
BINS 4(fp), 2
```

replaces bit 2 at address 0x0404 with a 1.



BNEZ *rx*, *offset*

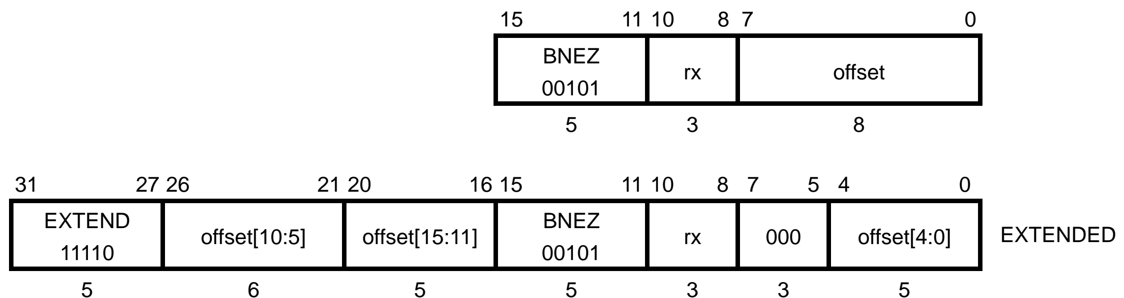
Branch On Not Equal To Zero

Operation

if $rx \neq 0$ then pc $\Leftarrow$ pc + 2 + sign-extend (*offset* $\|$ 0)

(EXTENDED) if $rx \neq 0$ then pc $\Leftarrow$ pc + 4 + sign-extend (*offset* $\|$ 0)

Instruction Encoding



Description

If the contents of general-purpose register *rx* is not equal to zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). See Section 5.3.4, *Branch Instructions (16-Bit ISA)*, for pipeline delays. This instruction does not have a delay slot. If the branch is taken, the instruction that immediately follows this instruction is not executed. The target address is computed relative to the address of the immediately following instruction, i.e., PC+2 when the instruction is not EXTENDED and PC+4 when EXTENDed.

With the 8-bit *offset* field, the branch range is -256 to +254. If the *offset* is outside this range, the instruction is EXTENDed to provide a 17-bit signed immediate in the range of -65536 to +65534. Whether EXTENDed or not, the target address is computed in the same manner.

Exceptions

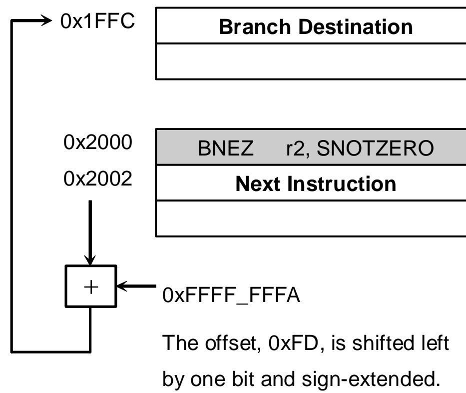
None

Example

```
BNEZ r2,SNOTZERO
```

Assume that this branch instruction resides at address 0x2000 and that label SNOTZERO points to absolute address 0x1FFC. Then the assembler/linker turns this label into an offset operand of 0xFD (see the figure below). Thus the instruction code for this branch instruction becomes 0x2AFD.

If the contents of r2 are not equal to zero, the processor transfers program control to address 0x1FFC. Otherwise, the program just continues to the next instruction at 0x2002.



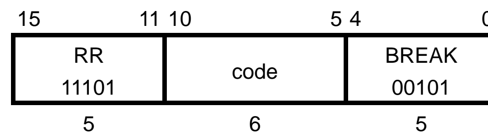
BREAK *code*

Breakpoint Exception

Operation

Breakpoint exception

Instruction Encoding



Description

When this instruction is executed, a breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.

The *code* field in the BREAK instruction is available for use as software parameters to pass additional information. The exception handler can retrieve it by loading the contents of the memory halfword containing the instruction. For more on this, see Section 9.1.11, *Breakpoint Exception*.

Exceptions

Breakpoint exception

BS1F *ry*, *rx*

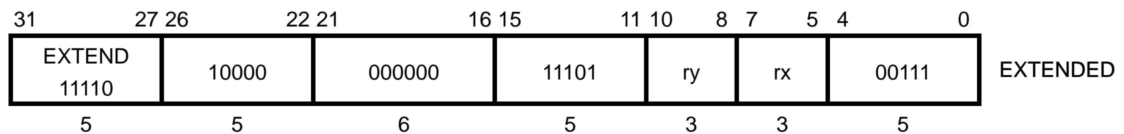
Bit Search One Forward

Operation

if $rx == 0$ then $ry \Leftarrow 0$;

else $ry \Leftarrow$ (bit position of rx[bit position] == 1) + 1 ;

Instruction Encoding



Description

General-purpose register *rx* is searched for the first set bit, starting from bit 0 towards bit 31. If a set bit is found in *rx*, its bit position (bit number plus 1) is placed into general-purpose register *ry*. If no set bit is found in *rx*, the value written to *ry* is 0.

Exceptions

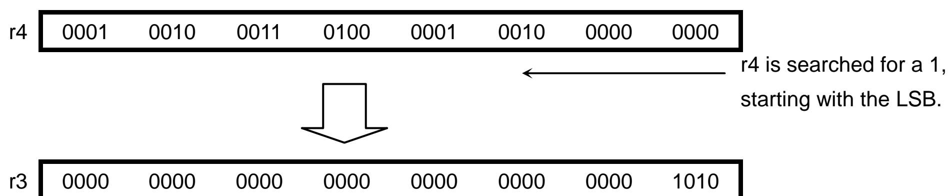
None

Example

Assume that general-purpose register r4 contains 0x1234_1200 (bit 9 is set). Then, the instruction:

```
BS1F  r3, r4
```

loads general-purpose register r3 with 0x0000_000A.



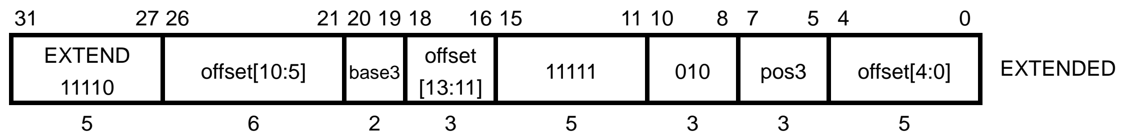
BSET *offset* (*base3*), *pos3*

Bit Set

Operation

{zero-extend (*offset*) + (*base3*)} [*pos3*] ⇐ 1

Instruction Encoding



Description

A bit specified by *pos3* in a memory byte is set. The effective address is computed by zero-extending the 14-bit *offset* and adding the resultant value to the contents of a general-purpose register indicated by *base3*. The encoding used for *base3* is as follows:

base3	GPR
01	gp(r28)
10	sp(r29)
11	fp(r30)

With the 14-bit *offset* field, the offset range is 0 to +16383.

Exceptions

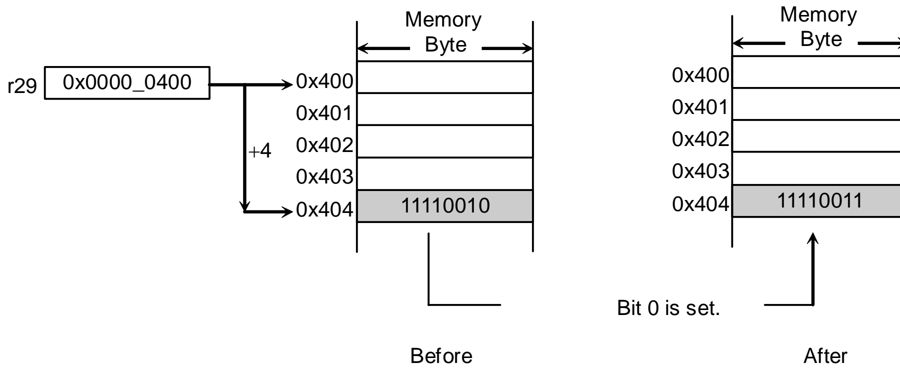
Address Error exception

Example

Assume that the sp register (r29) contains 0x0000_0400 and that the byte position at address 0x0404 contains 0xF2. Then, the instruction:

```
BSET 4(sp), 0
```

sets bit 0 of byte data 0xF2 as shown below.



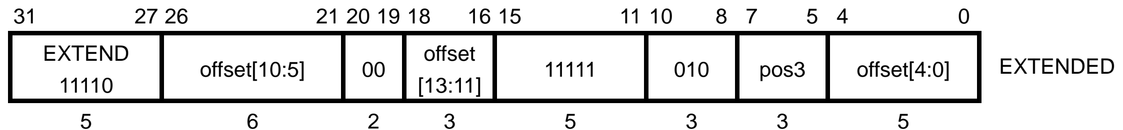
BSET *offset* (r0), *pos3*

Bit Set

Operation

{sign-extend (*offset*)} [*pos3*] ⇐ 1

Instruction Encoding



Description

A bit specified by *pos3* in a memory byte is negated and placed into the least-significant bit (LSB) of general-purpose register t8 (r24). The upper 31 bits of t8 are filled with zeros. The effective address is computed by sign-extending the 14-bit *offset* and adding the resultant value to the contents of general-purpose r0, which is hardwired to a value of zero.

With the 14-bit *offset* field, the offset range is -8192 to +8191.

Exceptions

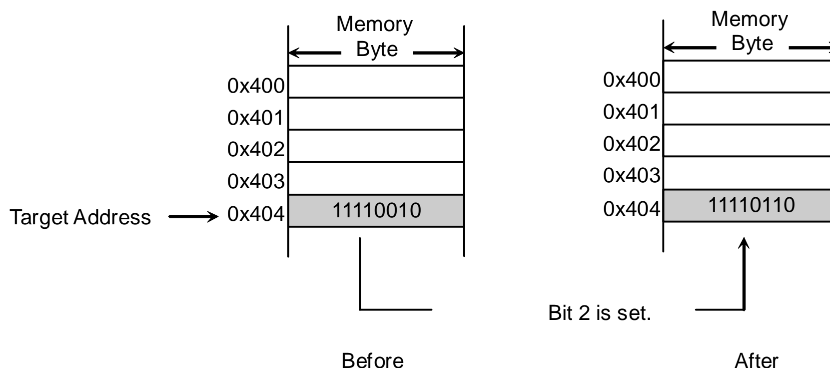
Address Error exception

Example

Assume that the byte position at address 0x0404 contains 0xF2. Then, the instruction:

```
BSET 0x404(r0), 2
```

sets bit 2 of byte data 0xF2 as shown below.



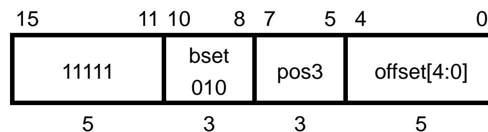
BSET *offset* (fp), *pos3*

Bit Set

Operation

{zero-extend (*offset*) + (fp)} [*pos3*] ⇐ 1

Instruction Encoding



Description

A bit specified by *pos3* in a memory byte is set. The effective address is computed by zero-extending the 5-bit *offset* and adding the resultant value to the contents of the fp register (r30).

With the 5-bit *offset* field, the offset range is 0 to +31.

Exceptions

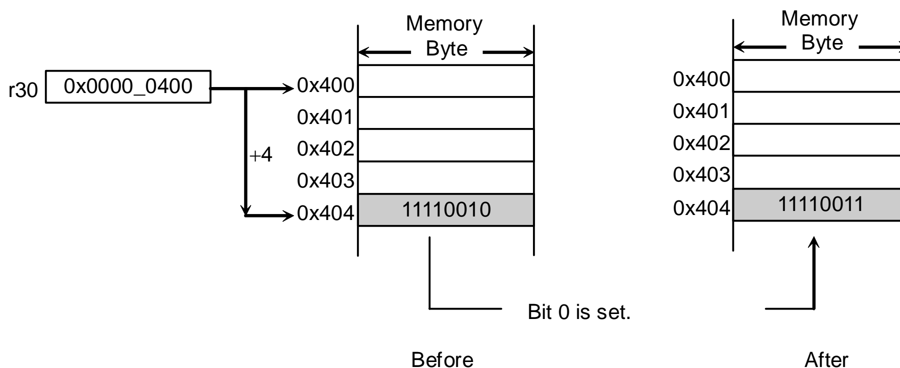
Address Error exception

Example

Assume that the fp register (r30) contains 0x0000_0400 and that the byte position at address 0x0404 contains 0xF2. Then, the instruction:

```
BSET 4(fp), 0
```

sets bit 0 of byte data 0xF2 as shown below.



BTEQZ *offset*

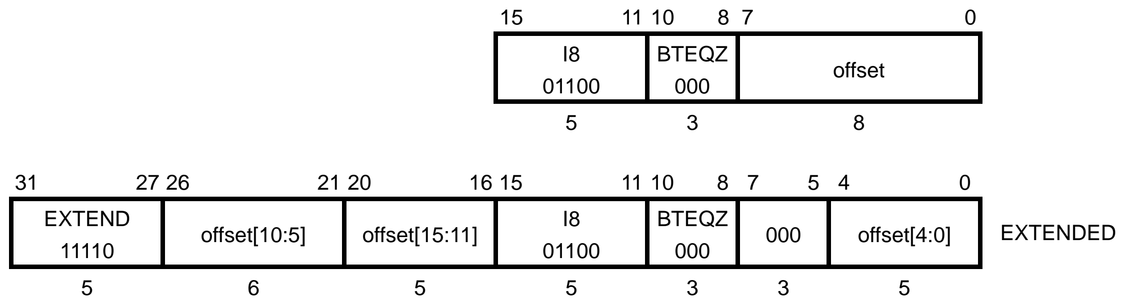
Branch On T8 Equal To Zero

Operation

if t8 == 0 then pc ⇐ pc + 2 + sign-extend (*offset* || 0)

(EXTENDED) if t8 == 0 then pc ⇐ pc + 4 + sign-extend (*offset* || 0)

Instruction Encoding



Description

If the contents of condition code register t8 (r24) is equal to zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). See Section 5.3.4, *Branch Instructions (16-Bit ISA)*, for pipeline delays. This instruction does not have a delay slot. If the branch is taken, the instruction that immediately follows this instruction is not executed. The target address is computed relative to the address of the immediately following instruction, i.e., PC+2 when the instruction is not EXTENDED and PC+4 when EXTENDed.

Since the 8-bit *offset* is shifted left by one bit, the branch range is -256 to +254. If the *offset* is outside this range, the instruction is EXTENDed to provide a 17-bit signed immediate in the range of -65536 to +65534. Whether EXTENDed or not, the target address is computed in the same manner.

Exceptions

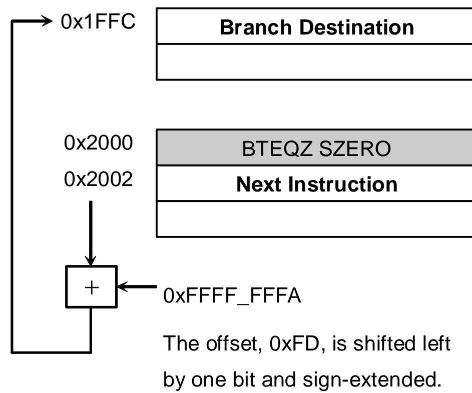
None

Example

```
BTEQZ SZERO
```

Assume that this branch instruction resides at address 0x2000 and that label SZERO points to absolute address 0x1FFC. Then the assembler/linker turns this label into an offset operand of 0xFD (see the figure below). Thus the instruction code for this branch instruction becomes 0x60FD.

If the contents of t8 are equal to zero, the processor transfers program control to address 0x1FFC. Otherwise, the program just continues to the next instruction at 0x2002.



BTNEZ *offset*

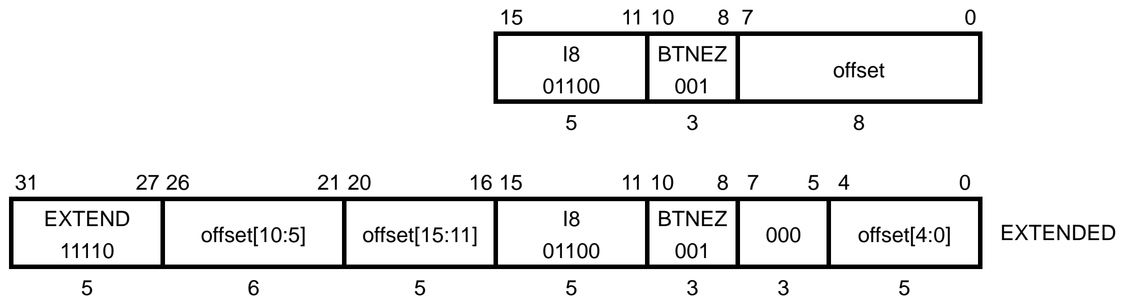
Branch On T8 Not Equal To Zero

Operation

if $t8 \neq 0$ then $pc \Leftarrow pc + 2 + \text{sign-extend}(offset \parallel 0)$

(EXTENDED) if $t8 \neq 0$ then $pc \Leftarrow pc + 4 + \text{sign-extend}(offset \parallel 0)$

Instruction Encoding



Description

If the contents of condition code register t8 (r24) is not equal to zero, then the program branches to the target address with a delay of one instruction (or two pipeline cycles). See Section 5.3.4, *Branch Instructions (16-Bit ISA)*, for pipeline delays. This instruction does not have a delay slot. If the branch is taken, the instruction that immediately follows this instruction is not executed. The target address is computed relative to the address of the immediately following instruction, i.e., PC+2 when the instruction is not EXTENDED and PC+4 when EXTENDed.

Since the 8-bit *offset* is shifted left by one bit, the branch range is -256 to +254. If the *offset* is outside this range, the instruction is EXTENDed to provide a 17-bit signed immediate in the range of -65536 to +65534. Whether EXTENDed or not, the target address is computed in the same manner.

Exceptions

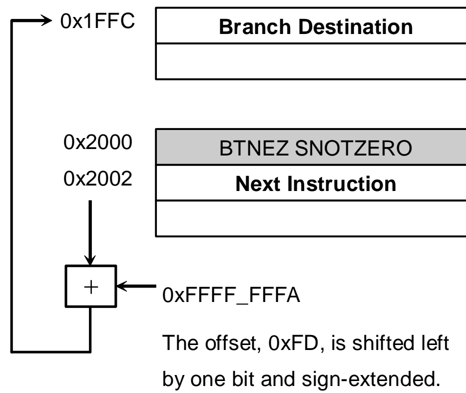
None

Example

```
BTNEZ SNOTZERO
```

Assume that this branch instruction resides at address 0x2000 and that label SNOTZERO points to absolute address 0x1FFC. Then the assembler/linker turns this label into an offset operand of 0xFD (see the figure below). Thus the instruction code for this branch instruction becomes 0x61FD.

If the contents of t8 are equal to zero, the processor transfers program control to address 0x1FFC. Otherwise, the program just continues to the next instruction at 0x2002.



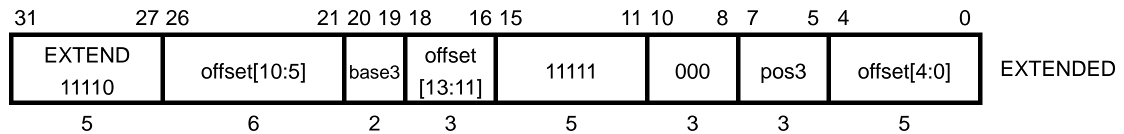
BTST *offset* (*base3*), *pos3*

Bit Test

Operation

t8 ⇐ 31’b 000_0000_0000_0000_0000_0000_0000_0000 || NOT ({zero-extend (*offset*) + (*base3*)} [*pos3*])

Instruction Encoding



Description

A bit specified by *pos3* in a memory byte is negated and placed into the least-significant bit (LSB) of general-purpose register t8 (r24). The upper 31 bits of t8 are filled with zeros. The effective address is computed by zero-extending the 14-bit *offset* and adding the resultant value to the contents of a general-purpose register indicated by *base3*. The encoding used for *base3* is as follows:

base3	GPR
01	gp (r28)
10	sp (r29)
11	fp (r30)

With the 14-bit *offset* field, the offset range is 0 to +16383.

Exceptions

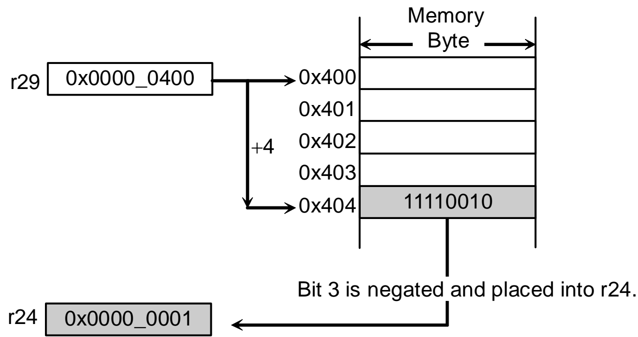
Address Error exception

Example

Assume that the sp register (r29) contains 0x0000_0400 and that the byte position at address 0x0404 contains 0xF2. Then, the instruction:

```
BTST 4(sp), 3
```

loads r24 with 0x0000_0001.



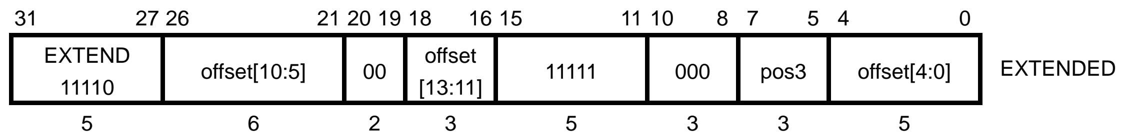
BTST *offset* (r0), *pos3*

Bit Test

Operation

t8 ⇐ 31’b 000_0000_0000_0000_0000_0000_0000_0000 || NOT ({sign-extend (*offset*)} [*pos3*])

Instruction Encoding



Description

A bit specified by *pos3* in a memory byte is negated and placed into the least-significant bit (LSB) of general-purpose register t8 (r24). The upper 31 bits of t8 are filled with zeros. The effective address is computed by sign-extending the 14-bit *offset* and adding the resultant value to the contents of general-purpose r0, which is hardwired to a value of zero.

With the 14-bit *offset* field, the offset range is -8192 to +8191.

Exceptions

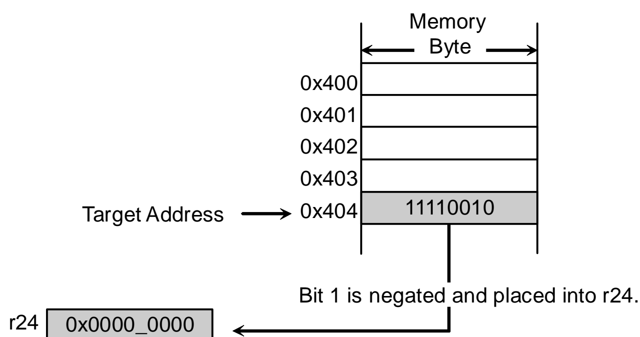
Address Error exception

Example

Assume that the byte position at address 0x0404 contains 0xF2. Then, the instruction:

```
BTST 0x404(r0), 1
```

loads r24 with 0x0000_0000.



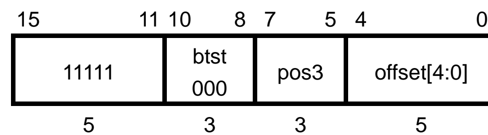
BTST *offset* (fp), *pos3*

Bit Test

Operation

t8 ⇐ 31’b 000_0000_0000_0000_0000_0000_0000_0000 || NOT ({zero-extend (*offset*) + (fp)} [*pos3*])

Instruction Encoding



Description

A bit specified by *pos3* in a memory byte is negated and placed into the least-significant bit (LSB) of general-purpose register t8 (r24). The upper 31 bits of t8 are filled with zeros. The effective address is computed by zero-extending the 5-bit *offset* and adding the resultant value to the contents of the fp register (r30).

With the 5-bit *offset* field, the offset range is 0 to +31.

Exceptions

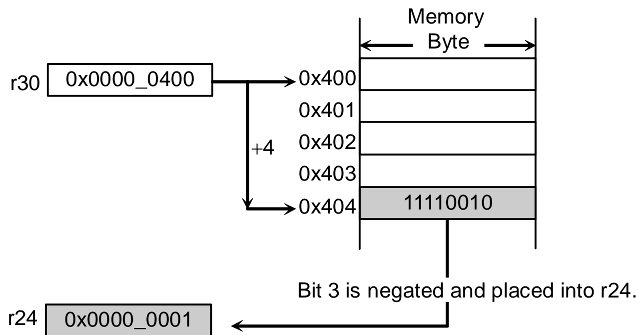
Address Error exception

Example

Assume that the fp register (r30) contains 0x0000_0400 and that the byte position at address 0x0404 contains 0xF2. Then, the instruction:

```
BTST 4(fp), 3
```

loads r24 with 0x0000_0001.



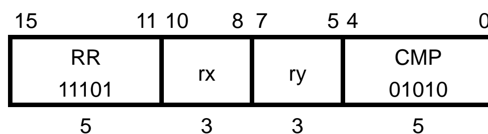
CMP *rx*, *ry*

Compare

Operation

if *rx* == *ry* then t8 ⇐ 0; else t8 ⇐ non-zero value

Instruction Encoding



Description

The contents of general-purpose register *rx* is exclusive-ORed with the contents of general-purpose register *ry*. The result is placed into condition code register t8 (r24). In other words, if *rx* and *ry* are equal, t8 is loaded with a value of zero.

Exceptions

None

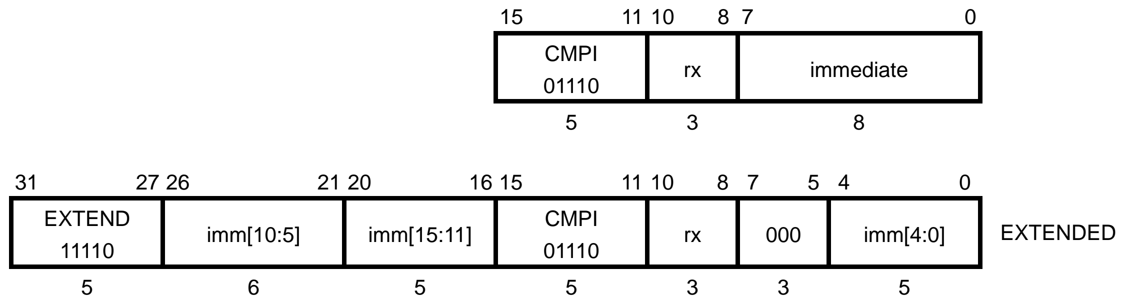
CMPI *rx*, *immediate*

Compare Immediate

Operation

if $rx == 0^{16} \parallel (immediate_{15..0})$ then t8 ⇐ 0; else t8 ⇐ non-zero value

Instruction Encoding



Description

The 8-bit *immediate* is zero-extended and exclusive-ORed with the contents of general-purpose register *rx*. The result is placed into condition code register t8 (r24). In other words, if *rx* and *immediate* are equal, t8 is loaded with a value of zero.

With the 8-bit *immediate* field, the immediate range is 0 to 255. If the *immediate* is larger than 255, the instruction is EXTENDed to provide a 16-bit unsigned immediate in the range of 0 to 65535.

Exceptions

None

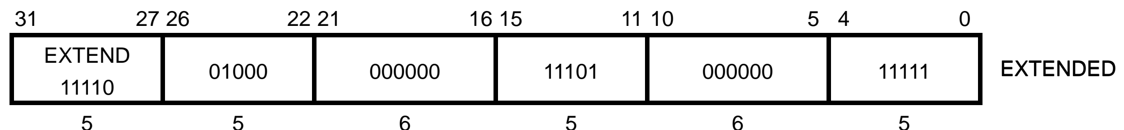
DERET

Debug Exception Return

Operation

pc ⇐ DEPC, Debug[DM] ⇐ 0, Debug[IEXI] ⇐ 0

Instruction Encoding



Description

The DERET instruction is used to return control from a debug exception handler to a user program. This is accomplished by loading the contents of the DEPC register into the program counter (PC). See Section 9.3.6, *Returning from Debug Exceptions*, for details.

The DERET instruction does not have a delay slot. It is executed with a delay of one instruction (or two pipeline cycles).

The DERET instruction restores the ISA mode bit (bit 0) of the PC from bit 0 of the DEPC register, bringing the processor into the ISA mode that had been in effect before the debug exception was taken.

The DERET instruction may not be in a jump or branch delay slot.

The operation of the DERET instruction is unpredictable if the processor is not in Debug mode (i.e., if the DM bit in the Debug register is cleared).

Typically, the DEPC register automatically captures the address of the exception-causing instruction on a debug exception. If you want to use the MTC0 instruction to load the DEPC register with a return address, the debug exception handler must execute at least two instructions before issuing the DERET instruction.

Exceptions

None

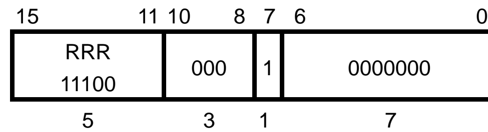
DI

Disable Interrupt

Operation

Status[IE] ⇐ 0

Instruction Encoding



Description

The IE bit in the Status register is cleared.

Exceptions

Coprocessor Unusable exception

DIV *rx*, *ry*

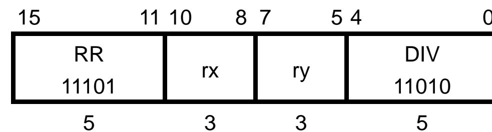
Divide

Operation

LO ⇐ *rx* ÷ *ry*;

HI ⇐ *rx* MOD *ry*

Instruction Encoding



Description

The contents of general-purpose register *rx* is divided by the contents of general-purpose register *ry*. Both operands are treated as signed integers. The quotient is placed into register LO and the remainder is placed into register HI. The DIV instruction never causes an Integer Overflow exception.

The result of the DIV instruction is undefined if the divisor is zero. Typically, it is necessary to check for a zero divisor and an overflow condition after a DIV instruction.

Any divide instruction is transferred to the dedicated divide unit as remaining instructions continue through the pipeline. The divide unit keeps running even when delay cycles and exceptions occur.

If the divide instruction is followed by an MFHI, MFLO, MADD, MADDU, MSUB or MSUBU instruction before the quotient and the remainder are available, the pipeline stalls until they do become available (see Section 5.5, *Divide Instructions*).

Exceptions

None

DIVE *rx*, *ry*

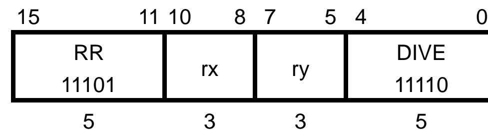
Divide Exception

Operation

LO ⇐ *rx* ÷ *ry*

HI ⇐ *rx* MOD *ry*

Instruction Encoding



Description

The contents of general-purpose register *rx* is divided by the contents of general-purpose register *ry*. Both operands are treated as signed integers. The quotient is placed into register LO and the remainder is placed into register HI.

An Integer Overflow exception occurs if divide-by-zero or overflow conditions are detected.

Any divide instruction is transferred to the dedicated divide unit as remaining instructions continue through the pipeline. The divide unit keeps running even when delay cycles and exceptions occur.

If the divide instruction is followed by an MFHI, MFLO, MADD, MADDU, MSUB or MSUBU instruction before the quotient and the remainder are available, the pipeline stalls until they do become available (see Section 5.5, *Divide Instructions*).

Exceptions

Integer Overflow exception

DIVEU *rx*, *ry*

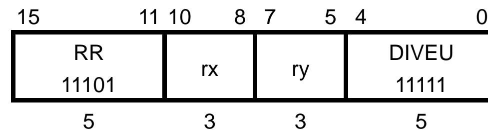
Divide Exception Unsigned

Operation

LO $\Leftarrow$ $rx \div ry$

HI $\Leftarrow$ rx MOD ry

Instruction Encoding



Description

The contents of general-purpose register *rx* is divided by the contents of general-purpose register *ry*. Both operands are treated as unsigned integers. The quotient is placed into register LO and the remainder is placed into register HI.

An Integer Overflow exception occurs if divide-by-zero is detected.

Any divide instruction is transferred to the dedicated divide unit as remaining instructions continue through the pipeline. The divide unit keeps running even when delay cycles and exceptions occur.

If the divide instruction is followed by an MFHI, MFLO, MADD, MADDU, MSUB or MSUBU instruction before the quotient and the remainder are available, the pipeline stalls until they do become available (see Section 5.5, *Divide Instructions*).

Exceptions

Integer Overflow exception

DIVU *rx*, *ry*

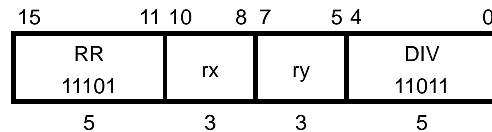
Divide Unsigned

Operation

LO ⇐ *rx* ÷ *ry*;

HI ⇐ *rx* MOD *ry*

Instruction Encoding



Description

The contents of general-purpose register *rx* is divided by the contents of general-purpose register *ry*. Both operands are treated as unsigned integers. The quotient is placed into register LO and the remainder is placed into register HI. The DIV instruction never causes Integer Overflow exceptions. The only difference between the DIV instruction and this instruction is that this instruction treats both operands as unsigned integers.

Exceptions

None

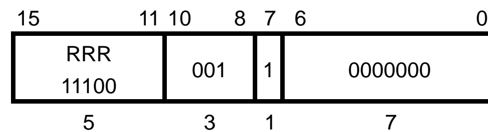
EI

Enable Interrupt

Operation

Status[IE] ⇐ 1

Instruction Encoding



Description

The IE bit in the Status register is set.

Exceptions

Coprocessor Unusable exception

ERET

Exception Return

Operation

if Status[ERL] = 1 then pc ⇐ Error EPC

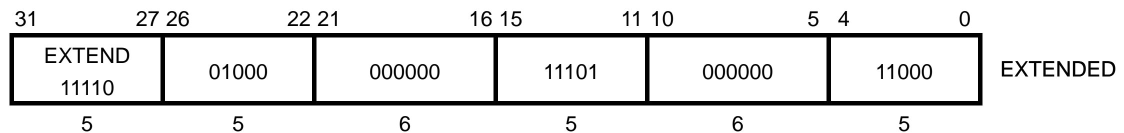
Status[ERL] ⇐ 0

else pc ⇐ EPC

Status[EXL] ⇐ 0

SSCR[CSS] ⇐ SSCR[PSS]

Instruction Encoding



Description

ERET is an instruction for returning from an interrupt, exception or error trap.

The ERET instruction does not have a delay slot. It is executed with a delay of one instruction (two pipeline cycles).

The ERET instruction restores the ISA mode bit (bit 0) of the PC from bit 0 of the ErrorEPC register, bringing the processor into the ISA mode that had been in effect before the exception was taken.

An attempt to execute the ERET instruction in User mode when the CU0 bit in the Status register is cleared causes a Coprocessor Unusable exception. If you want to use the MTC0 instruction to load the ErrorEPC or EPC register with a return address or if you have modified the contents of the Status register, the exception handler must execute at least two instructions before issuing the ERET instruction.

If the ERL bit in the Status register is set, ERET restores the PC from the ErrorPC register and then clears the ERL bit. Otherwise, ERET restores the PC from the EPC register and then clears the EXL bit.

Also, the PSS field in the SSCR register is popped to the CSS field.

ERET must not be placed in a branch or jump delay slot.

Exceptions

Coprocessor Unusable exception

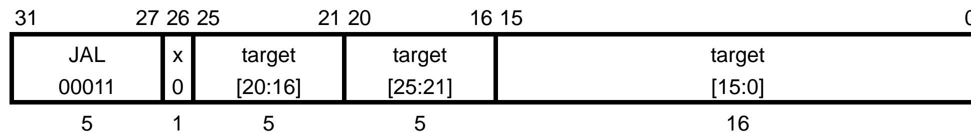
JAL *target*

Jump And Link

Operation

ra ⇐ pc + 7; pc ⇐ pc[31:28] || *target* || 00

Instruction Encoding



Description

Although this instruction is in the 16-bit ISA, it is 32-bits wide. The program unconditionally jumps to the target address with a delay of one instruction (or two pipeline cycles). See Section 5.3.3, *Jump Instructions (16-Bit ISA)*. The target address is computed relative to the address of the instruction in the jump delay slot (PC+4). The 26-bit *target* is shifted left by two bits and combined with the four most-significant bits of PC+4 to form the target address. The JAL instruction never toggles the ISA mode bit of the program counter (PC).

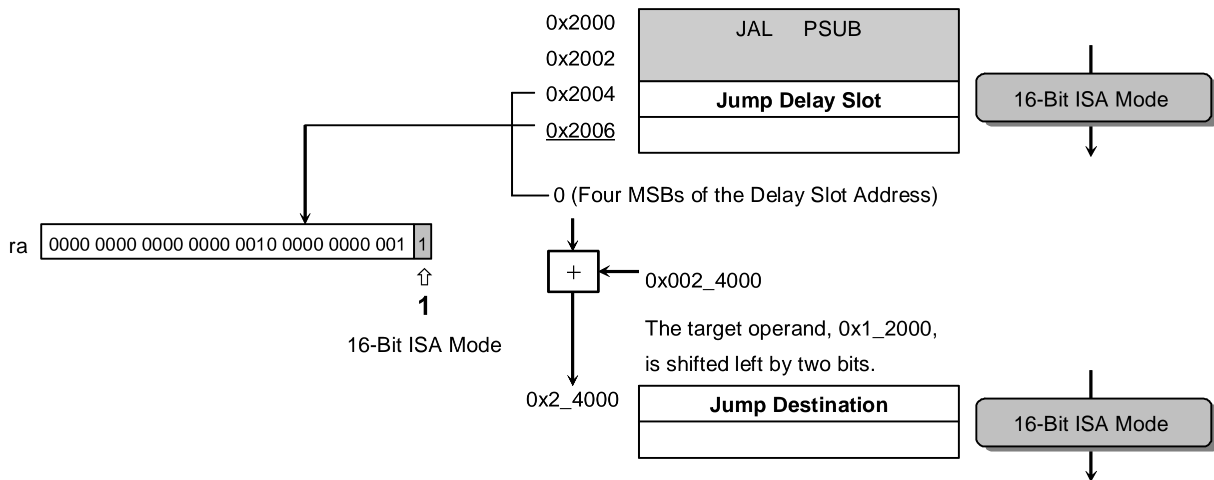
The address of the instruction after the jump delay slot is saved in the link register, ra (r31). The ISA mode specifier (i.e., a 1 for the 16-bit ISA mode) is saved in the least-significant bit of ra.

Example

```
JAL PSUB
```

Assume that this jump instruction resides at address 0x2000 and that label PSUB points to absolute address 0x2_4000. Then the assembler/linker turns this label into a target operand of 0x1_2000 (see the figure below).

The processor unconditionally transfers program control to address 0x2_4000. The jump takes effect after the instruction in the jump delay slot is executed. The address of the instruction after the jump delay slot is saved in ra, combined with the ISA mode bit value; thus the ra value becomes 0x0000_2007.



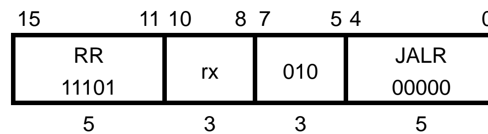
JALR ra, *rx*

Jump And Link Register

Operation

ra ⇐ pc + 5; pc ⇐ *rx*

Instruction Encoding



Description

The program unconditionally jumps to the address contained in general-purpose register *rx*, with the least-significant bit cleared, with a delay of one instruction (or two pipeline cycles). The least-significant bit of *rx* is interpreted as the ISA mode specifier. The address of the instruction after the jump delay slot is saved in the link register, ra (r31), combined with the value of the ISA mode that was in effect before the jump.

In 32-bit ISA mode, all instructions must be aligned on word boundaries. Therefore, when jumping to a 32-bit routine, the two low-order bits of the target register (*rx*) must be zero. If the two low-order bits are not zero, an Address Error exception will occur when the processor fetches the instruction at the jump destination.

Exceptions

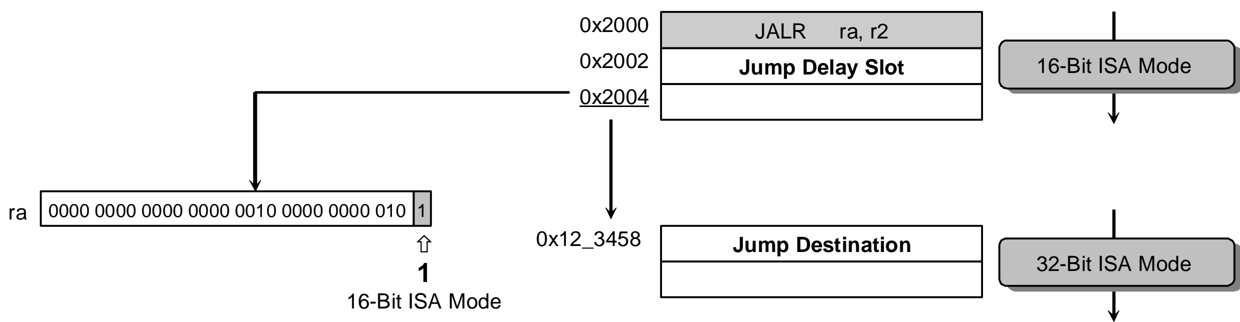
None

Example

Assume that register r2 contains 0x0012_3458 and that the following jump instruction resides at address 0x0000_2000. Then, executing the instruction:

```
JALR ra,r2
```

transfers program control to address 0x0012_3458. Since r2 has the least-significant bit cleared, the ISA mode bit toggles to 0 after the jump, bringing the processor into 32-bit ISA mode. The address of the instruction after the jump delay slot is saved in ra, combined with the ISA mode bit value; thus the ra value becomes 0x0000_2005.



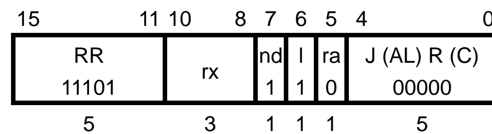
JALRC ra, *rx*

Jump And Link Register, Compact

Operation

ra ⇐ pc + 3; pc ⇐ *rx*

Instruction Encoding



Description

The program unconditionally jumps to the address contained in general-purpose register *rx*, with the least-significant bit cleared, with a delay of one instruction (or two pipeline cycles). This instruction does not have a delay slot; the address of the instruction following this instruction is saved in the link register, ra (r31), combined with the ISA mode bit.

In 32-bit ISA mode, all instructions must be aligned on word boundaries. Therefore, when jumping to a 32-bit routine, the two low-order bits of the target register (*rx*) must be zero. If the two low order bits are not zero, an Address Error exception will occur when the processor fetches the instruction at the jump destination.

Exceptions

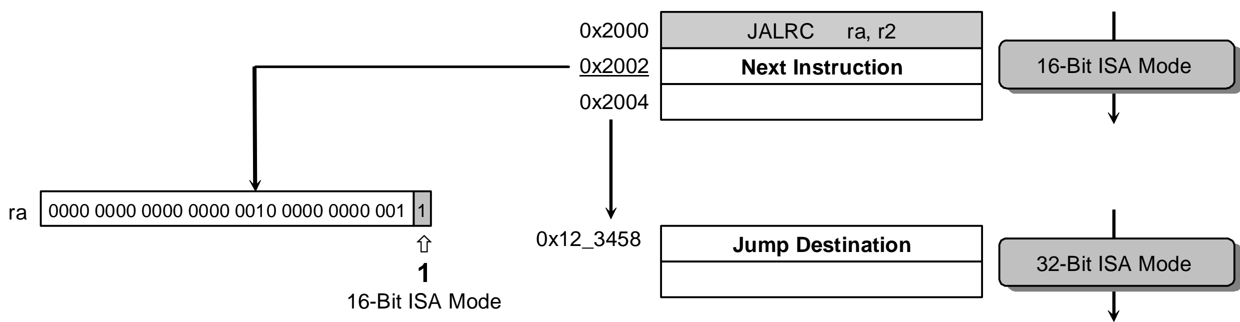
None

Example

Assume that register r2 contains 0x0012_3458 and that the following jump instruction resides at address 0x0000_2000. Then, executing the instruction:

```
JALRC ra,r2
```

transfers program control to address 0x0012_3458. Since r2 has the least-significant bit cleared, the ISA mode bit toggles to 0 after the jump, bringing the processor into 32-bit ISA mode. The address of the instruction after this instruction is saved in ra, combined with the ISA mode bit value; thus the ra value becomes 0x0000_2003.



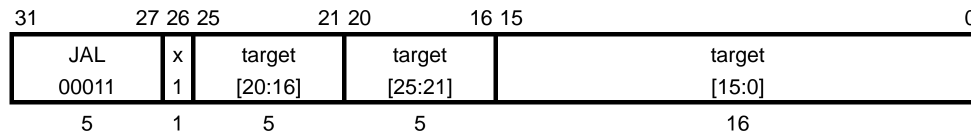
JALX *target*

Jump And Link eXchange

Operation

ra ⇐ pc + 7; pc[31:1] ⇐ pc[31:28] || *target* || 00; pc[0] ⇐ NOT pc[0]

Instruction Encoding



Description

Although this instruction is in the 16-bit ISA, it is 32-bits wide. The program unconditionally jumps to the target address with a delay of one instruction (or two pipeline cycles). See Section 5.3.3, *Jump Instructions (16-Bit ISA)*. The target address is computed relative to the address of the instruction in the jump delay slot (PC+4). The 26-bit target is shifted left by two bits and combined with the four most-significant bits of PC+4 to form the target address. The JALX instruction unconditionally toggles the ISA mode bit of the program counter (PC).

The address of the instruction after the jump delay slot is saved in the link register, ra (r31). The least-significant bit of ra stores the ISA mode bit that was in effect before the jump.

Exceptions

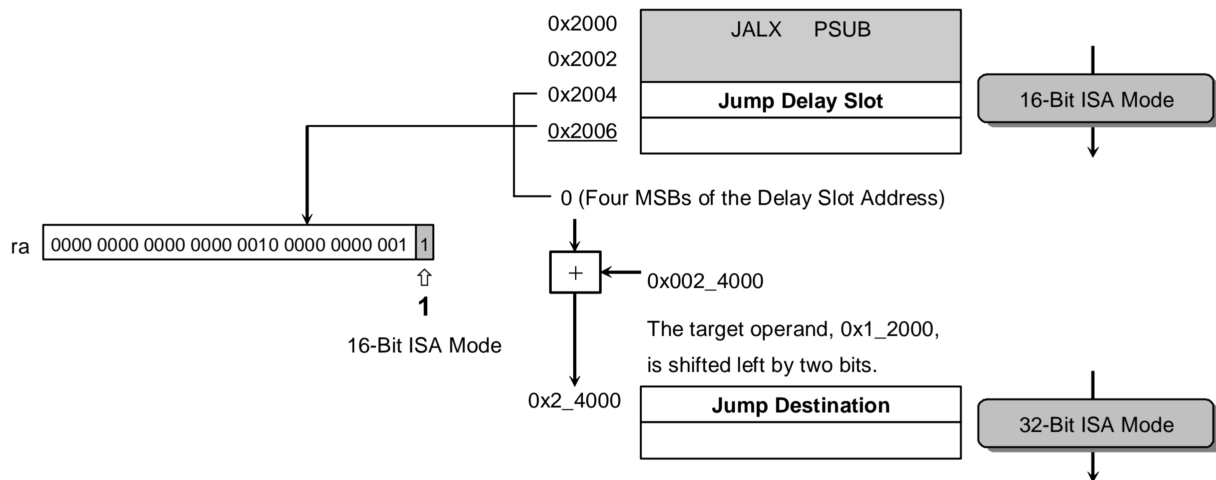
None

Example

```
JALX PSUB
```

Assume that this jump instruction resides at address 0x0000_2000 and that label PSUB points to absolute address 0x2_4000. Then, the assembler/linker turns this label into a target operand of 0x1_2000 (see the figure below).

The processor unconditionally transfers program control to address 0x2_4000. The jump takes effect after the instruction in the jump delay slot is executed. The ISA mode bit unconditionally toggles, bringing the processor into 32-bit ISA mode. The address of the instruction after the jump delay slot is saved in ra, combined with the ISA mode bit value; thus the ra value becomes 0x0000_2007.



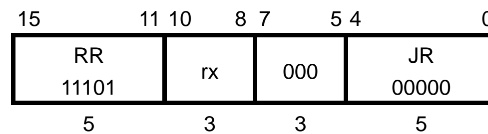
JR *rx*

Jump Register

Operation

pc ⇐ *rx*

Instruction Encoding



Description

The program unconditionally jumps to the address contained in general-purpose register *rx*, with the least-significant bit cleared, with a delay of one instruction (or two pipeline cycles). The least-significant bit of *rx* is interpreted as the ISA mode specifier.

In 32-bit ISA mode, all instructions must be aligned on word boundaries. Therefore, when jumping to a 32-bit routine, the two low-order bits of the target register (*rx*) must be zero. If the two low-order bits are not zero, an Address Error exception will occur when the processor fetches the instruction at the jump destination.

Exceptions

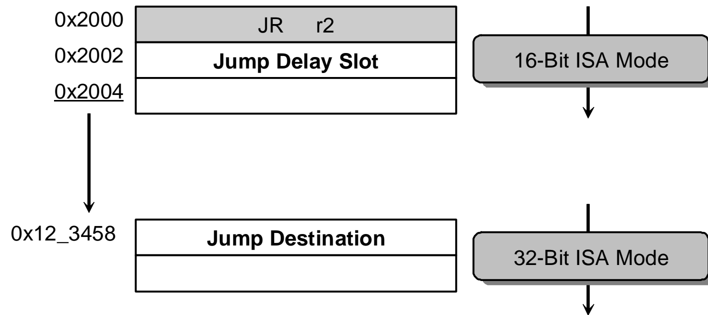
None

Example

Assume that register r2 contains 0x0012_3458. Then, executing the instruction:

```
JR r2
```

transfers program control to address 0x0012_3458. Since r2 has the least-significant bit cleared, the processor switches to 32-bit ISA mode. The jump takes effect after the instruction in the jump delay slot is executed.



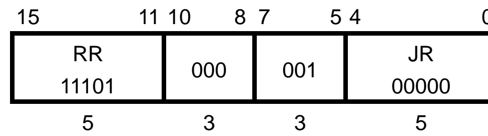
JR ra

Jump Register

Operation

pc ⇐ ra

Instruction Encoding



Description

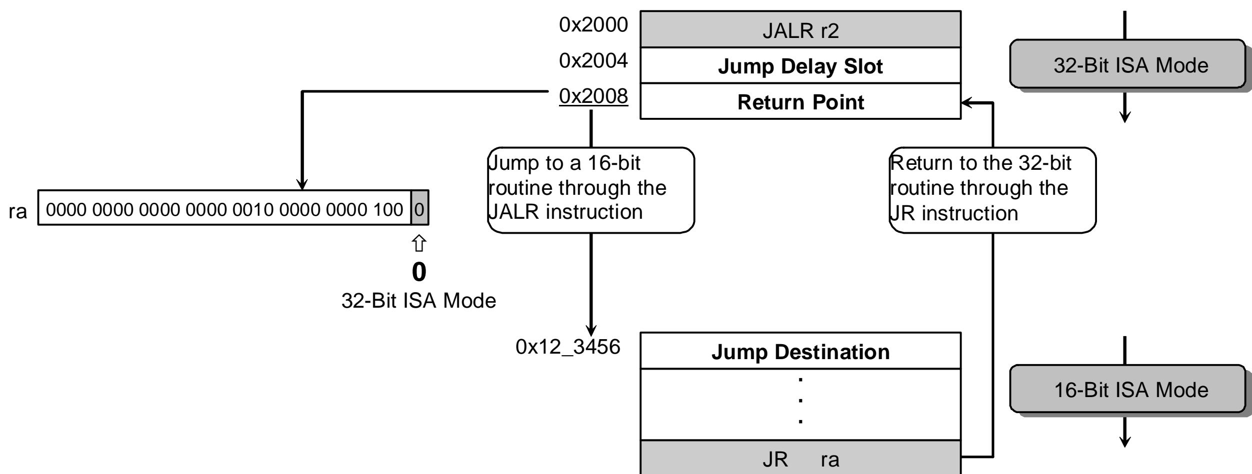
The program unconditionally jumps to the address contained in the link register, ra (r31), with the least-significant bit cleared, with a delay of one instruction (or two pipeline cycles). The least-significant bit of ra is interpreted as the ISA mode specifier.

Exceptions

None

Example

In the following example, the JALR instruction in a 32-bit routine transfers program control to a 16-bit routine. At the end of the 16-bit routine, the JR instruction restores the return address into the program counter (PC) from the link register, ra (r31). Since the ISA mode has been saved in the least-significant bit of ra by the 32-bit JALR instruction, executing the JR instruction at the end of the 16-bit routine restores it into the PC, causing the processor to revert to 32-bit ISA mode.



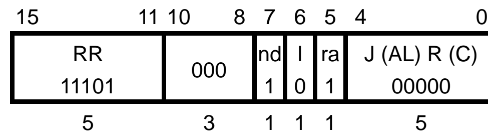
JRC ra

Jump Register ra, Compact

Operation

pc ⇐ ra

Instruction Encoding



Description

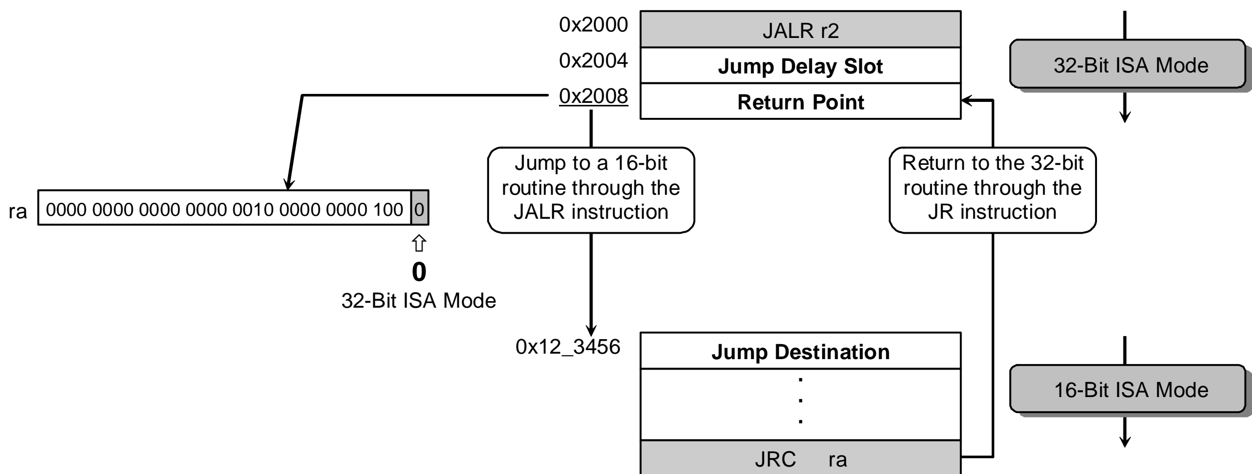
The program unconditionally jumps to the address contained in the link register, ra (r31), with the least-significant bit cleared, with a delay of one instruction (or two pipeline cycles). The least-significant bit of ra is interpreted as the ISA mode specifier.
This instruction does not have a delay slot.

Exceptions

None

Example

In the following example, the JALR instruction in a 32-bit routine transfers program control to a 16-bit routine. At the end of the 16-bit routine, the JRC instruction restores the return address into the program counter (PC) from the link register, ra (r31). Since the ISA mode has been saved in the least-significant bit of ra by the 32-bit JALR instruction, executing the JRC instruction at the end of the 16-bit routine restores it into the PC, causing the processor to revert to 32-bit ISA mode.



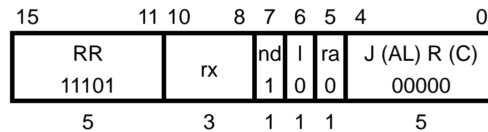
JRC *rx*

Jump Register, Compact

Operation

pc ⇐ *rx*

Instruction Encoding



Description

The program unconditionally jumps to the address contained in general-purpose register *rx*, with the least-significant bit cleared, with a delay of one instruction (or two pipeline cycles). The least-significant bit of *rx* is interpreted as the ISA mode specifier.

This instruction does not have a delay slot.

In 32-bit ISA mode, all instructions must be aligned on word boundaries. Therefore, when jumping to a 32-bit routine, the two low-order bits of the target register (*rx*) must be zero. If the two low-order bits are not zero, an Address Error exception will occur when the processor fetches the instruction at the jump destination.

Exceptions

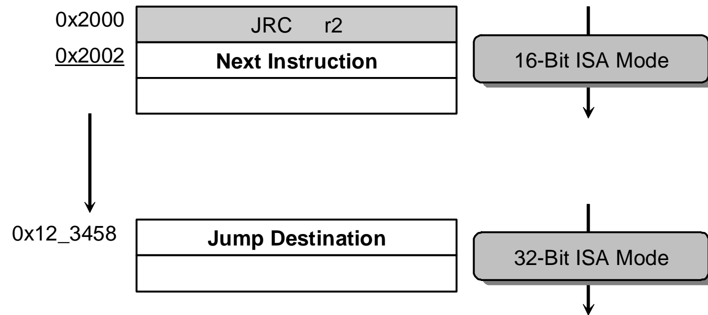
None

Example

Assume that register r2 contains 0x0012_3458. Then, executing the instruction:

```
JRC r2
```

transfers program control to address 0x0012_3458. Since r2 has the least-significant bit cleared, the ISA mode bit toggles to 0 after the jump, bringing the processor into 32-bit ISA mode. The instruction following this instruction is not executed.



LB *ry*, *offset* (*base*)

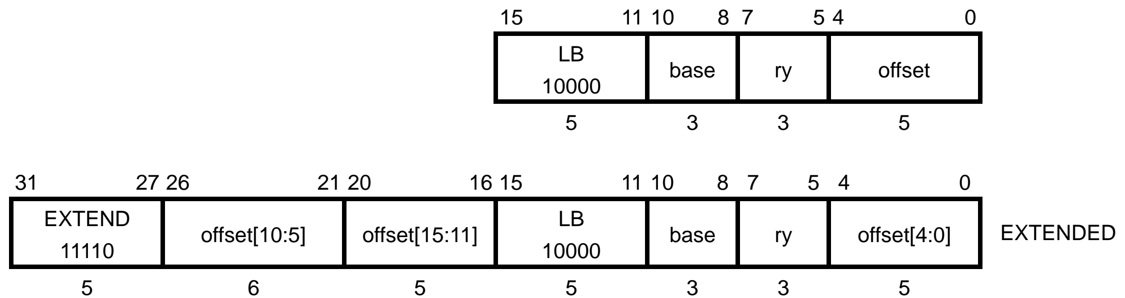
Load Byte

Operation

ry = {zero-extend (*offset*) + (*base*)}

(EXTENDED) *ry* = {sign-extend (*offset*) + (*base*)}

Instruction Encoding



Description

The 5-bit immediate *offset* is zero-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The byte in memory addressed by the EA is sign-extended and loaded into general-purpose register *ry*.

With the 5-bit *offset* field, the offset range is 0 to 31. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

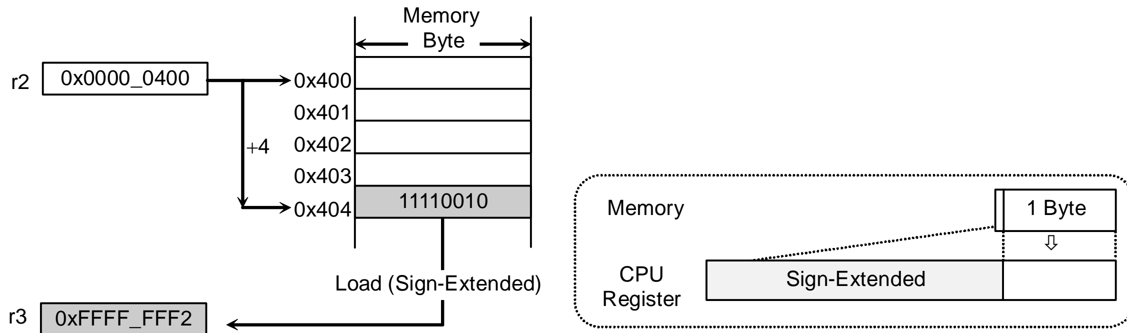
Address Error exception

Example

Assume that register r2 contains 0x0000_0400 and that the memory location at address 0x404 contains 0xF2. Then, executing the instruction:

```
LB r3,4(r2)
```

loads register r3 with 0xFFFF_FFF2.



LBU *ry*, *offset* (*base*)

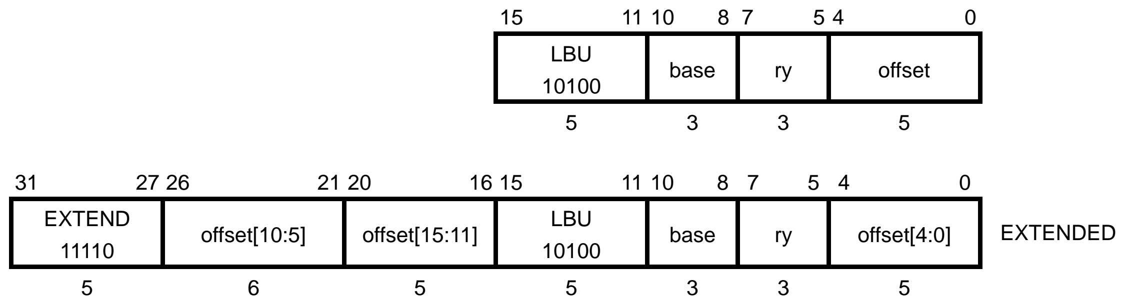
Load Byte Unsigned

Operation

ry = {zero-extend (*offset*) + (*base*)}

(EXTENDED) *ry* = {sign-extend (*offset*) + (*base*)}

Instruction Encoding



Description

The 5-bit immediate *offset* is zero-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The byte in memory addressed by the EA is zero-extended and loaded into general-purpose register *ry*.

With the 5-bit *offset* field, the offset range is 0 to 31. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

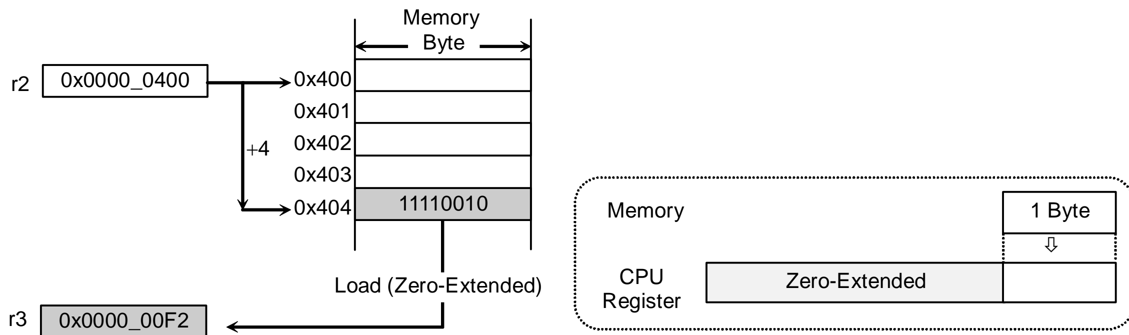
Address Error exception

Example

Assume that register r2 contains 0x0000_0400 and that the memory location at address 0x404 contains 0xF2. Then, executing the instruction:

```
LBU r3,4(r2)
```

loads register r3 with 0x0000_00F2.



LBU *ry*, *offset* (fp)

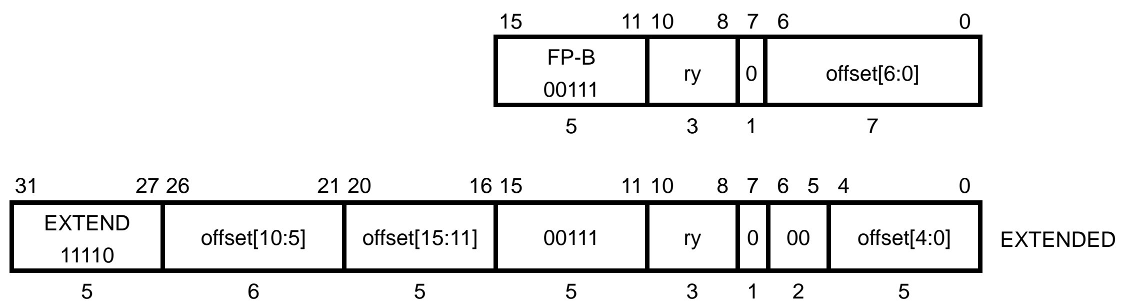
Load Byte Unsigned

Operation

ry = {zero-extend (*offset*) + (fp)}

(EXTENDED) *ry* = {sign-extend (*offset*) + (fp)}

Instruction Encoding



Description

The 7-bit immediate *offset* is zero-extended and added to the contents of the fp register (r30) to form an effective address (EA). The byte in memory addressed by the EA is zero-extended and loaded into general-purpose register *ry*.

With the 7-bit *offset* field, the offset range is 0 to 127. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

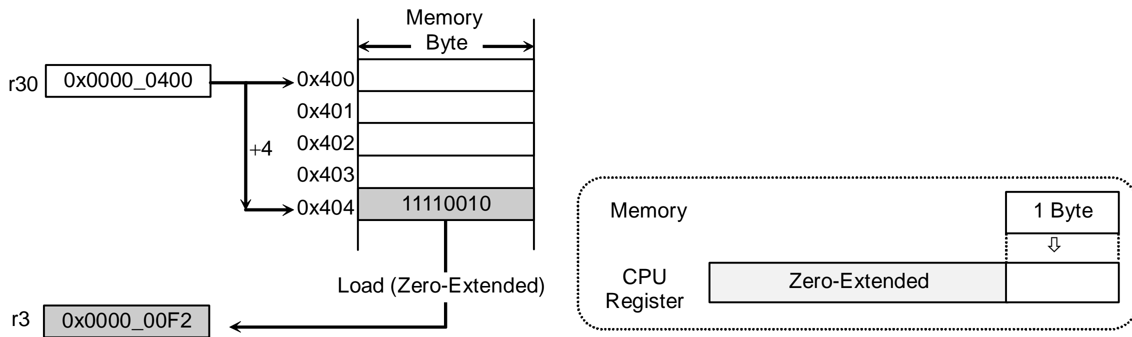
Address Error exception

Example

Assume that fp register (r30) contains 0x0000_0400 and that the memory location at address 0x404 contains 0xF2. Then, executing the instruction:

```
LBU r3,4(fp)
```

loads register r3 with 0x0000_00F2.



LBU *ry*, *offset* (sp)

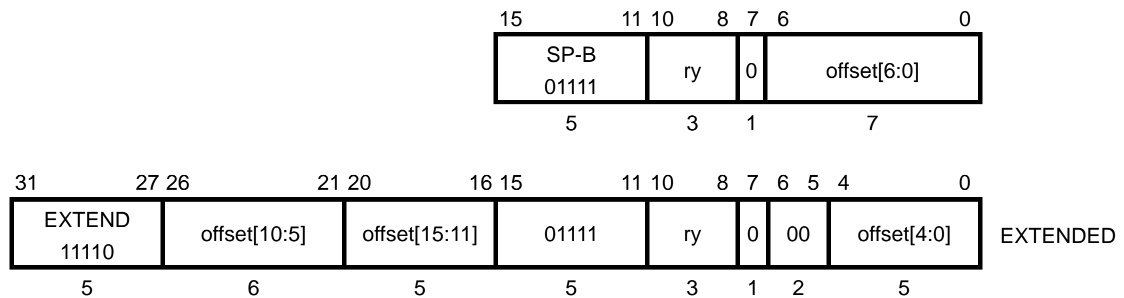
Load Byte Unsigned

Operation

ry = {zero-extend (*offset*) + (sp)}

(EXTENDED) *ry* = {sign-extend (*offset*) + (sp)}

Instruction Encoding



Description

The 7-bit immediate *offset* is zero-extended and added to the contents of the sp register (r29) to form an effective address (EA). The byte in memory addressed by the EA is zero-extended and loaded into general-purpose register *ry*.

With the 7-bit *offset* field, the offset range is 0 to 127. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

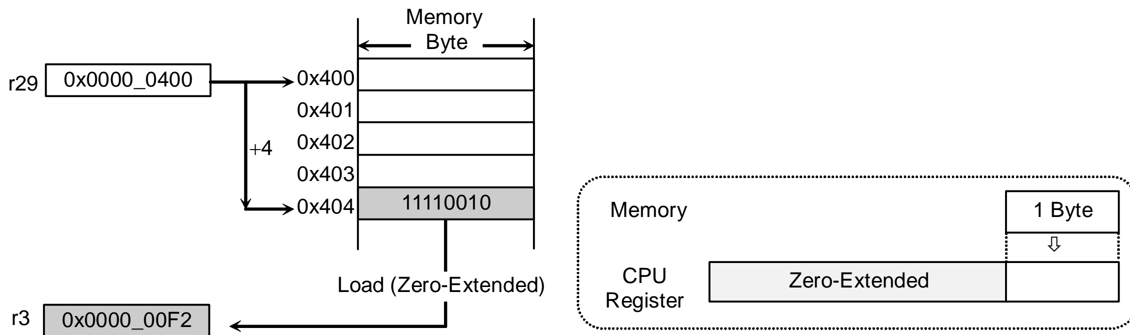
Address Error exception

Example

Assume that sp register (r29) contains 0x0000_0400 and that the memory location at address x404 contains 0xF2. Then, executing the instruction:

```
LBU r3,4(sp)
```

loads register r3 with 0x0000_00F2.



LH *ry*, *offset* (*base*)

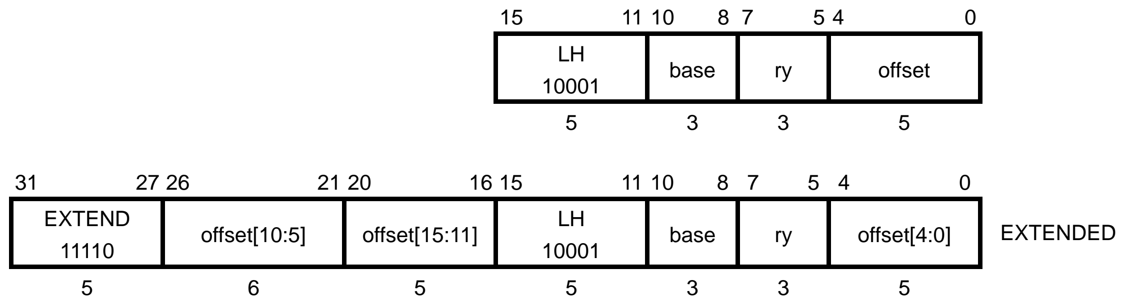
Load Halfword

Operation

$ry \Leftarrow \{\text{zero-extend } (offset \parallel 0) + (base)\}$

(EXTENDED) $ry \Leftarrow \{\text{sign-extend } (offset) + (base)\}$

Instruction Encoding



Description

The 5-bit immediate *offset* is shifted left by one bit, zero-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The halfword in memory addressed by the EA is sign-extended and loaded into general-purpose register *ry*.

Since the 5-bit *offset* is shifted left by one bit, the offset range is 0 to 62, in increments of two. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767. When EXTENDed, the *offset* operand is not shifted at all.

Exceptions

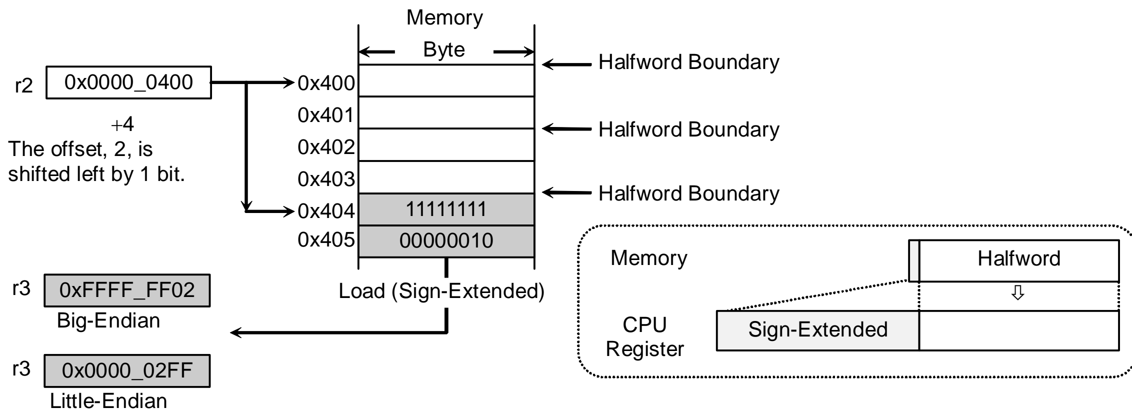
Address Error exception

Example

```
LH r3,4(r2)
```

Assume that register r2 contains 0x0000_0400 and that the memory locations at addresses 0x404 and 0x405 contain 0xFF and 0x02 respectively. Since the offset value is shifted left by one bit by the processor hardware, the assembler/linker turns the specified offset (4 or binary 0100) into a code of 2 (binary 0010). Thus the instruction code for this load instruction becomes 0x8A62.

This load instruction loads register r3 with 0xFFFF_FF02 in big-endian mode and with 0x0000_02FF in little-endian mode.



LHU *ry*, *offset* (*base*)

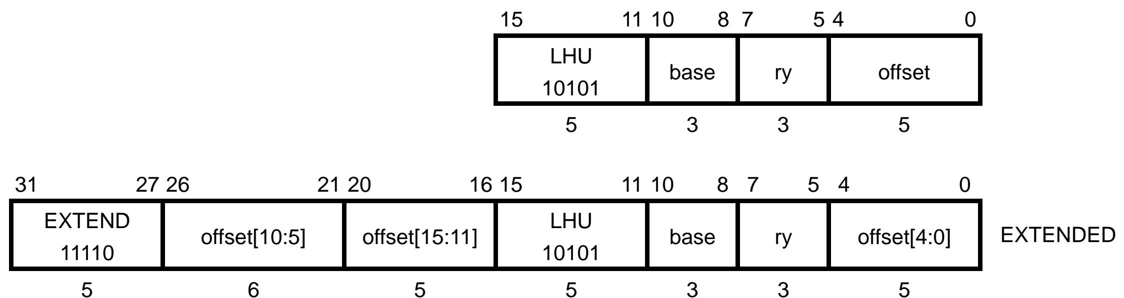
Load Halfword Unsigned

Operation

$ry \Leftarrow \{\text{zero-extend}\ (offset\ \|\ 0) + (base)\}$

(EXTENDED) $ry \Leftarrow \{\text{sign-extend}\ (offset) + (base)\}$

Instruction Encoding



Description

The 5-bit immediate *offset* is shifted left by one bit, zero-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The halfword in memory addressed by the EA is zero-extended and loaded into general-purpose register *ry*.

Since the 5-bit *offset* is shifted left by one bit, the offset range is 0 to 62, in increments of two. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767. When EXTENDed, the *offset* operand is not shifted at all.

Exceptions

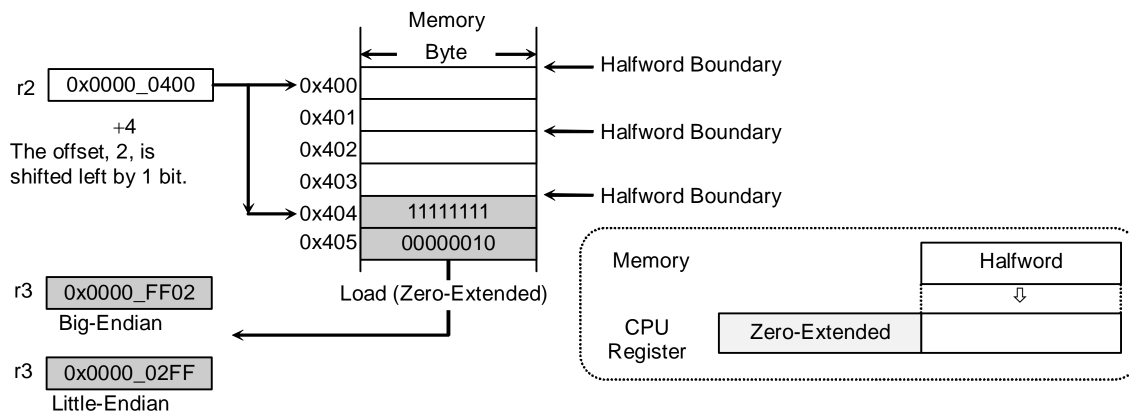
Address Error exception

Example

```
LHU r3,4(r2)
```

Assume that register r2 contains 0x0000_0400 and that the memory locations at addresses 0x404 and 0x405 contain 0xFF and 0x02 respectively. Since the offset value is shifted left by one bit by the processor hardware, the assembler/linker turns the specified offset (4 or binary 0100) into a code of 2 (binary 0010). Thus the instruction code for this load instruction becomes 0xAA62.

This load instruction loads register r3 with 0x0000_FF02 in big-endian mode and with 0x0000_02FF in little-endian mode.



LHU *ry*, *offset* (fp)

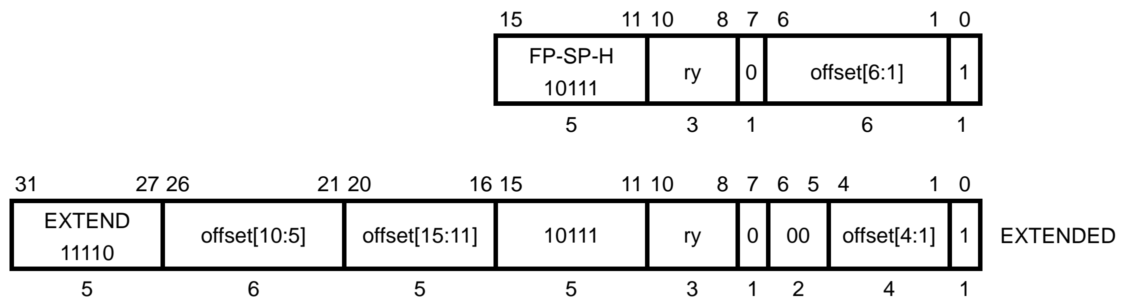
Load Halfword Unsigned

Operation

$ry \Leftarrow \{\text{zero-extend } (offset \parallel 0) + (fp)\}$

(EXTENDED) $ry \Leftarrow \{\text{sign-extend } (offset \parallel 0) + (fp)\}$

Instruction Encoding



Description

The 6-bit immediate *offset* is shifted left by one bit, zero-extended and added to the contents of the fp register (r30) to form an effective address (EA). The halfword in memory addressed by the EA is zero-extended and loaded into general-purpose register *ry*.

Since the 6-bit *offset* is shifted left by one bit, the offset range is 0 to 126, in increments of two. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate. When EXTENDed, the *offset* operand is shifted left by one bit to allow an offset of -32768 to +32766.

Exceptions

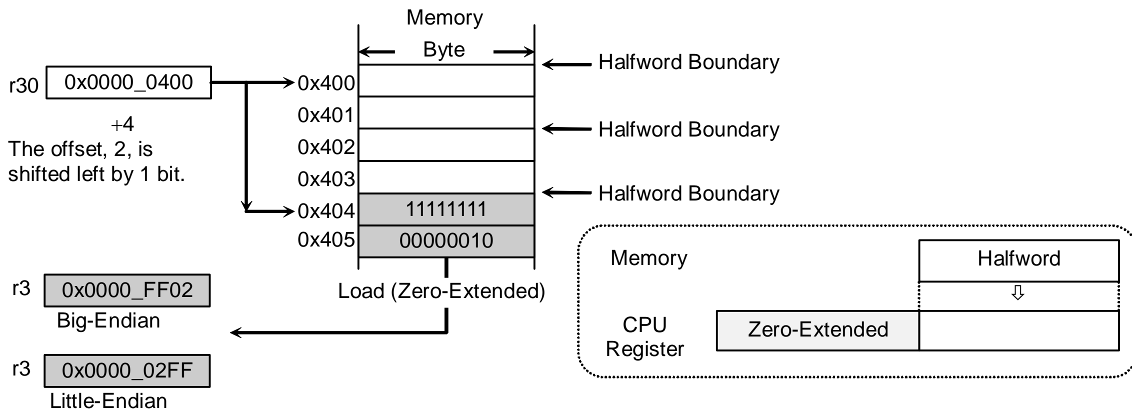
Address Error exception

Example

```
LHU r3,4(fp)
```

Assume that the fp register (r30) contains 0x0000_0400 and that the memory locations at addresses 0x404 and 0x405 contain 0xFF and 0x02 respectively. Since the offset value is shifted left by one bit by the processor hardware, the assembler/linker turns the specified offset (4 or binary 0100) into a code of 2 (binary 0010). Thus the instruction code for this load instruction becomes 0xBB05.

This load instruction loads register r3 with 0x0000_FF02 in big-endian mode and with 0x0000_02FF in little-endian mode.



LHU *ry*, *offset* (sp)

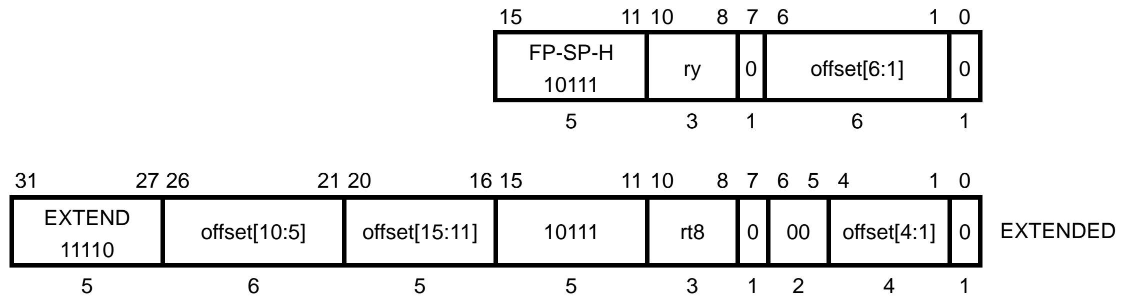
Load Halfword Unsigned

Operation

$ry \Leftarrow \{\text{zero-extend}\ (offset\ \|\ 0) + (\text{sp})\}$

(EXTENDED) $ry \Leftarrow \{\text{sign-extend}\ (offset\ \|\ 0) + (\text{sp})\}$

Instruction Encoding



Description

The 6-bit immediate *offset* is shifted left by one bit, zero-extended and added to the contents of the sp register (r29) to form an effective address (EA). The halfword in memory addressed by the EA is zero-extended and loaded into general-purpose register *ry*.

Since the 6-bit *offset* is shifted left by one bit, the offset range is 0 to 126, in increments of two. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate. When EXTENDed, the *offset* operand is shifted left by one bit to allow an offset of -32768 to +32766.

Exceptions

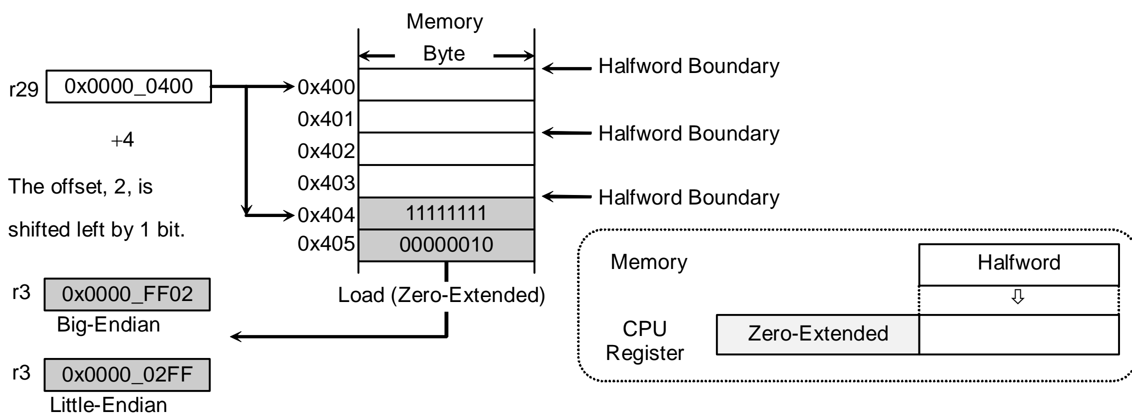
Address Error exception

Example

```
LHU r3,4(sp)
```

Assume that the sp register (r29) contains 0x0000_0400 and that the memory locations at addresses 0x404 and 0x405 contain 0xFF and 0x02 respectively. Since the offset value is shifted left by one bit by the processor hardware, the assembler/linker turns the specified offset (4 or binary 0100) into a code of 2 (binary 0010). Thus the instruction code for this load instruction becomes 0xBB04.

This load instruction loads register r3 with 0x0000_FF02 in big-endian mode and with 0x0000_02FF in little-endian mode.



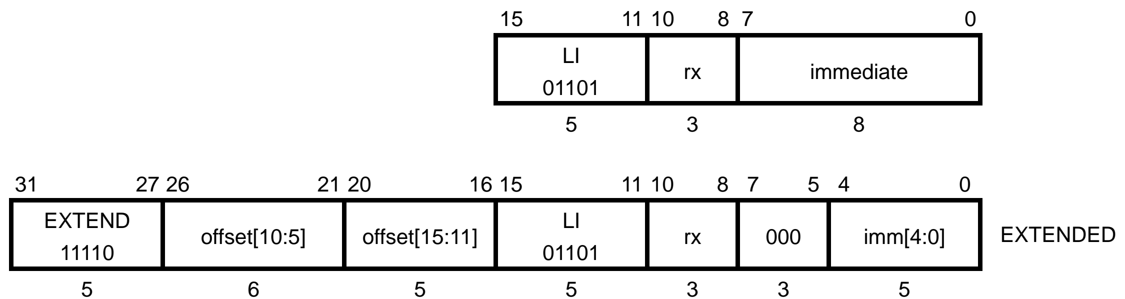
LI *rx*, *immediate*

Load Immediate

Operation

$rx \Leftarrow 0^{16} \,||\, (immediate_{15..0})$

Instruction Encoding



Description

The 8-bit *immediate* is zero-extended and loaded into general-purpose register *rx*.

With the 8-bit *immediate* field, the immediate range is 0 to 255. If the *immediate* is outside this range, the instruction is EXTENDed to provide a 16-bit unsigned immediate in the range of 0 to 65535.

Exceptions

None

Example

The instruction:

```
LI r3,0x12
```

loads register r3 with 0x0000_0012.

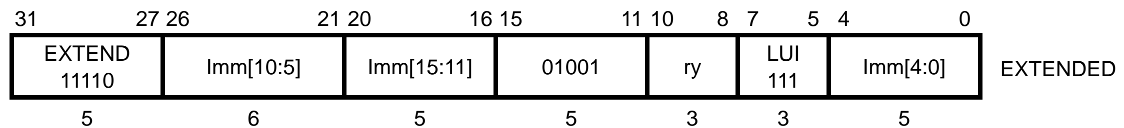
LUI *ry*, *immediate*

Load Upper Immediate

Operation

$ry \Leftarrow immediate \parallel 0x0000$

Instruction Encoding



Description

The 16-bit *immediate* is shifted left by 16 bits and concatenated to 16 bits of zeros. The result is placed into general-purpose register *ry*.

Exceptions

None

Example

The instruction:

```
LUI r4,0x1234
```

loads register r4 with 0x1234_0000.

LW *rx*, *offset* (pc)

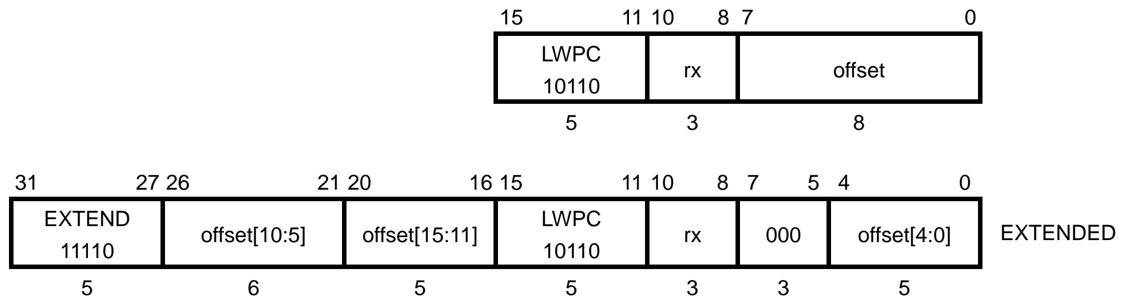
Load Word

Operation

$rx \Leftarrow \{\text{zero-extend}\ (offset\ \|\ 00) + (\text{Masked Base PC})\}$

(EXTENDED) $rx \Leftarrow \{\text{sign-extend}\ (offset) + (\text{Masked Base PC})\}$

Instruction Encoding



Description

The 8-bit immediate *offset* is shifted left by two bits, zero-extended and added to the contents of the program counter (PC) with the lower two bits cleared to form an effective address (EA). A 32-bit constant in memory addressed by the EA is then loaded into general-purpose register *rx*.

By virtue of this instruction, 32-bit constants can be embedded in the code segment. The LW instructions within the nearby routines can reference this data with a single instruction.

Since the 8-bit *offset* is shifted left by two bits, the offset range is 0 to 1020, in increments of four. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767. Given the PC-relative addressing mode, there is also an instruction (ADDIUPC) to calculate a PC-relative address and place it in a general-purpose register.

Because the PC value is used as the base value, it is commonly referred to as the base PC value. The base PC value with the lower two bits cleared is referred to as the masked base PC value. The base PC value varies, depending on whether the instruction is in a delay slot and whether it is to be EXTENDed.

LWPC	Base PC Value
Delay slot of the JR or JALR instruction	Address of the JR or JALR instruction
Delay slot of the JAL or JALX instruction	Address of the upper halfword of the JAL or JALX instruction
EXTENDed	Address of the EXTEND code
Not EXTENDed (nor in a delay slot)	Address of the LWPC instruction

Exceptions

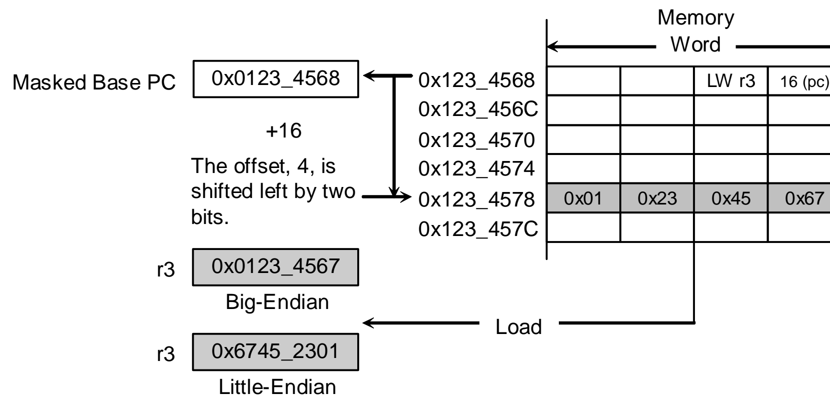
Address error exception

Example

Assume that the masked base PC points at address 0x0123_4568 and that addresses 0x1234_5678 to 0x0123_457B contain 0x01, 0x23, 0x45 and 0x67 respectively. Given the instruction:

```
LW r3,16(pc)
```

the assembler turns the specified offset value (16 or binary 0001_0000) into a code of 4 (binary 0000_ 0100) since it is to be shifted left by two bits by the processor hardware. Thus the instruction code for the above load instruction becomes 0xB304. Executing the above instruction loads register r3 with 0x0123_4567 in big-endian mode and with 0x6745_2301 in little-endian mode.



LW *rx*, *offset* (sp)

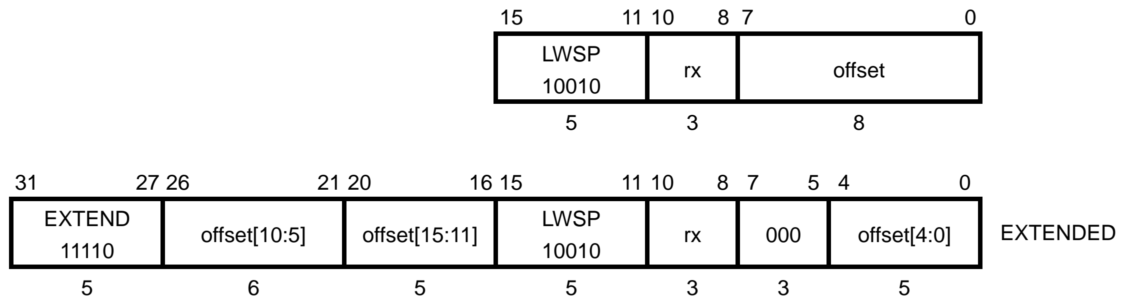
Load Word

Operation

$rx \Leftarrow \{\text{zero-extend}\ (offset \parallel 00) + (sp)\}$

(EXTENDED) $rx \Leftarrow \{\text{sign-extend}\ (offset) + (sp)\}$

Instruction Encoding



Description

The 8-bit immediate *offset* is shifted left by two bits, zero-extended and added to the contents of stack pointer register sp (r29) to form an effective address (EA). The word in memory addressed by the EA is loaded into general-purpose register *rx*.

Since the 8-bit *offset* is shifted left by two bits, the offset range is 0 to 1020, in increments of four. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

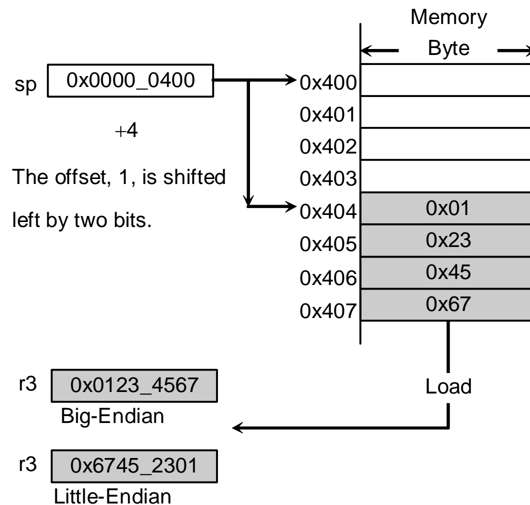
Address Error exception

Example

Assume that stack pointer register sp points at address 0x0000_0400 and that addresses 0x404 to 0x407 contain 0x01, 0x23, 0x45 and 0x67 respectively. Given the instruction:

```
LW r3,4(sp)
```

the assembler/linker turns the specified offset value (4 or binary 0100) into a code of 1 (binary 0001) since it is to be shifted left by two bits by the processor hardware. Thus the instruction code for the above load instruction becomes 0x9301. Executing the above instruction loads register r3 with 0x0123_4567 in big-endian mode and with 0x6745_2301 in little-endian mode.



LW *ry*, *offset* (*base*)

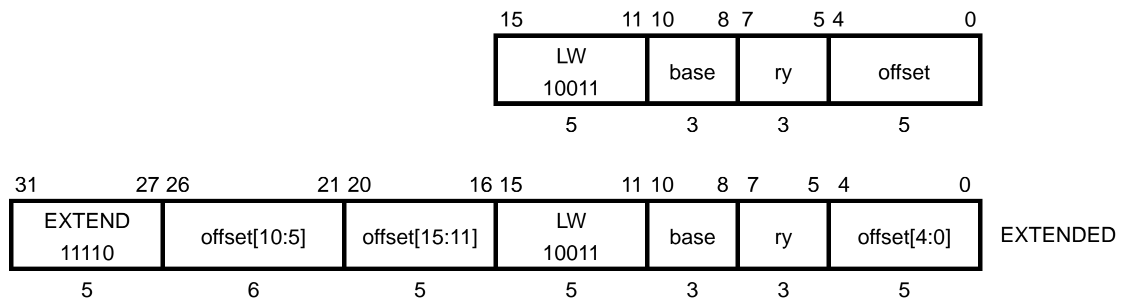
Load Word

Operation

$ry \Leftarrow \{\text{zero-extend}\ (offset \parallel 00) + (base)\}$

(EXTENDED) $ry \Leftarrow \{\text{sign-extend}\ (offset) + (base)\}$

Instruction Encoding



Description

The 5-bit immediate *offset* is shifted left by two bits, zero-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The word in memory addressed by the EA is loaded into general-purpose register *ry*.

Since the 5-bit *offset* is shifted left by two bits, the offset range is 0 to 124, in increments of four. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767. When EXTENDed, the *offset* operand is not shifted at all.

Exceptions

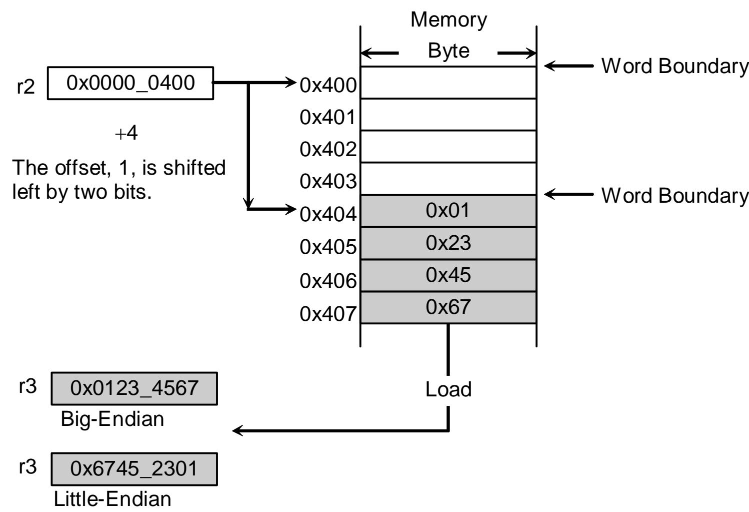
Address Error exception

Example

```
LW r3,4(r2)
```

Assume that register r2 contains 0x0000_0400 and that the memory locations at addresses 0x404 to 0x407 contain 0x01, 0x23, 0x45 and 0x67 respectively. Since the offset value is shifted left by two bits by the processor hardware, the assembler/linker turns the specified offset (4 or binary 0100) into a code of 1 (binary 0001). Thus the instruction code for this load instruction becomes 0x9A61.

This load instruction loads register r3 with 0x0123_4567 in big-endian mode and with 0x6745_2301 in little-endian mode.



LW *ry*, *offset* (fp)

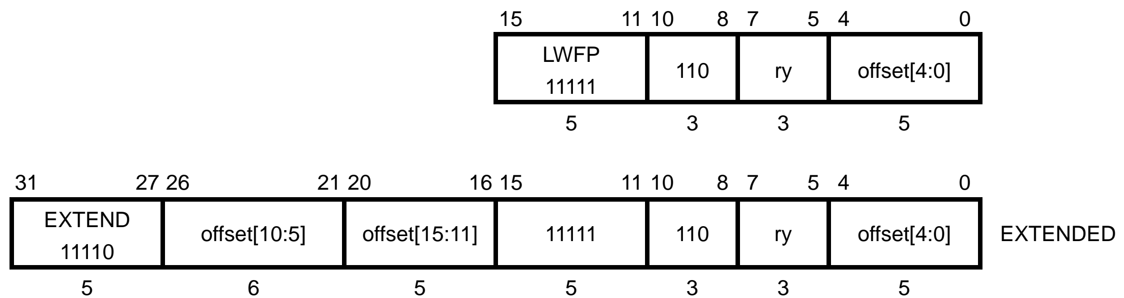
Load Word

Operation

$ry \Leftarrow \{\text{zero-extend } (offset \parallel 00) + (\text{fp})\}$

(EXTENDED) $ry \Leftarrow \{\text{sign-extend } (offset) + (\text{fp})\}$

Instruction Encoding



Description

The 5-bit immediate *offset* is shifted left by two bits, zero-extended and added to the contents of the fp register (r30) to form an effective address (EA). The word in memory addressed by the EA is loaded into general-purpose register *ry*.

Since the 5-bit *offset* is shifted left by two bits, the offset range is 0 to 124, in increments of four. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

Address Error exception

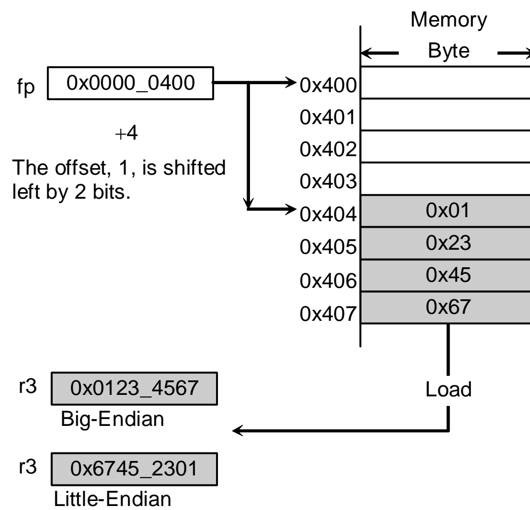
Example

Assume that the fp register (r30) points at address 0x0000_0400 and that addresses 0x404 to 0x407 contain 0x01, 0x23, 0x45 and 0x67 respectively. Given the instruction:

```
LW r3,4(fp)
```

the assembler/linker turns the specified offset value (4 or binary 0100) into a code of 1 (binary 0001) since it is to be shifted left by two bits by the processor hardware. Thus the instruction code for the above load instruction becomes 0xFE61.

Executing the above instruction loads register r3 with 0x0123_4567 in big-endian mode and with 0x6745_2301 in little-endian mode.



MADD *rx*, *ry*

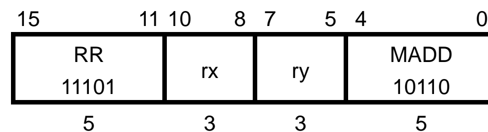
Multiply and Add

Operation

HI ⇐ high-order word of (HI || LO) + ($rx \times ry$)

LO ⇐ low-order word of (HI || LO) + ($rx \times ry$) の下位ワード

Instruction Encoding



Description

The contents of general-purpose register *rx* is multiplied by the contents of general-purpose register *ry*, and then the product is added to the 64-bit, doubleword contents of the HI and LO registers. Both *rx* and *ry* are treated as signed integers. The high-order word of the result is placed into the HI register, and the low-order word of the result is placed into the LO register.

No Integer Overflow exception occurs under any circumstances.

Exceptions

None

Example

Assume that the HI and LO registers contain 0x0000_0000 and 0xFFFF_FFFF respectively and that general-purpose registers r3 and r4 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MADD r3,r4
```

evaluates:

0x0000_0000_FFFF_FFFF + (0x0123_4567 · 0x89AB_CDEF)
= 0x0000_0000_FFFF_FFFF + 0xFF79_5E36_C94E_4629
= 0xFF79_5E37_C94E_4628

Hence, the high-order word of the result, 0xFF79_5E37, is placed into the HI register, and the low-order word of the result, 0xC94E_4628, is placed into the LO register.

MADDU *rx, ry*

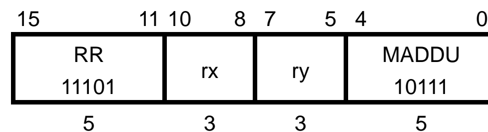
Multiply and Add Unsigned

Operation

HI $\Leftarrow$ high-order word of (HI $\|$ LO) + ($rx \times ry$)

LO $\Leftarrow$ low-order word of (HI $\|$ LO) + ($rx \times ry$)

Instruction Encoding



Description

The contents of general-purpose register *rx* is multiplied by the contents of general-purpose register *ry*, and then the product is added to the 64-bit, doubleword contents of the HI and LO registers. Both *rx* and *ry* are treated as unsigned integers. The high-order word of the result is placed into the HI register, and the low-order word of the result is placed into the LO register.

No Integer Overflow exception occurs under any circumstances.

Exceptions

None

Example

Assume that the HI and LO registers contain 0x0000_0000 and 0xFFFF_FFFF respectively and that general-purpose registers r3 and r4 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MADDU r3,r4
```

evaluates:

0x0000_0000_FFFF_FFFF + (0x0123_4567 · 0x89AB_CDEF)
= 0x0000_0000_FFFF_FFFF + 0x009C_A39D_C94E_4629
= 0x009C_A39E_C94E_4628

Hence, the high-order word of the result, 0x009C_A39E, is placed into the HI register, and the low-order word of the result, 0xC94E_4628, is placed into the LO register.

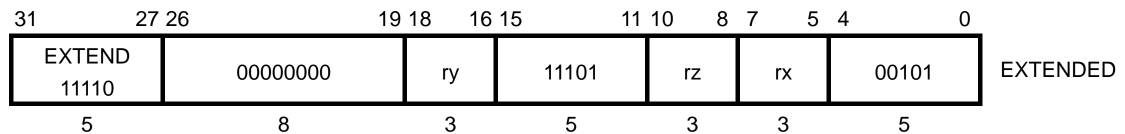
MAX *rz, rx, ry*

Maximum Signed

Operation

$$\text{if } rx > ry \text{ then } rz \Leftarrow rx;$$
$$\text{else } rz \Leftarrow ry;$$

Instruction Encoding



Description

The contents of general-purpose register *rx* is compared to the contents of general-purpose register *ry* as signed values. If *rx* is greater than *ry*, the value of *rx* is written to general-purpose register *rz*. Otherwise, the value of *ry* is written to *rz*.

Exceptions

None

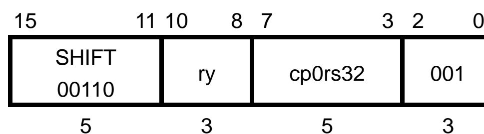
MFC0 *ry*, *cp0rs32*

Move from Coprocessor 0

Operation

ry ⇐ CP0 register *cp0rs32*

Instruction Encoding



Description

The contents of CP0 register *cp0rs32* is loaded into general-purpose register *ry*.
In 16-bit ISA mode, this instruction can not access the Config1, Config2, Config3 and IER registers.

Exceptions

Coprocessor Unusable exception

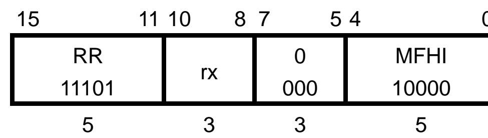
MFHI *rx*

Move From HI

Operation

rx ⇐ HI

Instruction Encoding



Description

The contents of the HI register is loaded into general-purpose register *rx*.

Exceptions

None

MFLO *rx*

Move From LO

Operation

$rx \Leftarrow \mathrm{LO}$

Instruction Encoding

15 — 11	10 — 8	7 — 5	4 — 0
RR 11101	rx	0 000	MFLO 10010
5	3	3	5

Description

The contents of the LO register is loaded into general-purpose register *rx*.

Exceptions

None

MIN *rz*, *rx*, *ry*

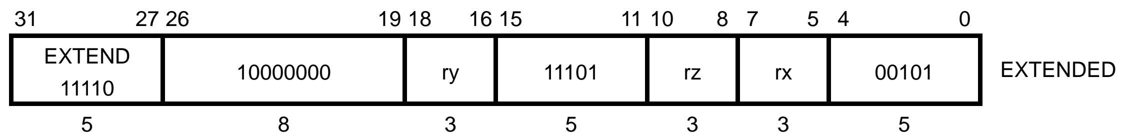
Minimum Signed

Operation

if $rx < ry$ then $rz \Leftarrow rx$;

else $rz \Leftarrow ry$;

Instruction Encoding



Description

The contents of general-purpose register *rx* is compared to the contents of general-purpose register *ry* as signed values. If *rx* is less than *ry*, the value of *rx* is written to general-purpose register *rz*. Otherwise, the value of *ry* is written to *rz*.

Exceptions

None

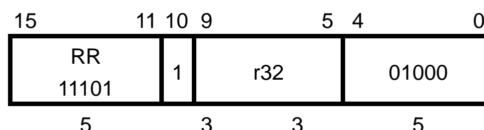
MOVE fp, *r32*

Move

Operation

fp ⇐ *r32*

Instruction Encoding



Description

The contents of general-purpose register *r32* is copied into the fp register (r30), where *r32* is any of the 32 registers (r0 to r31).

The encoding of the *r32* field in the 16-bit instruction code is as follows.

Code	Register
00000	r0
00001	r1
00010	r2
00011	r3
00100	r4
00101	r5
00110	r6
00111	r7
01000	r8
01001	r9
01010	r10
01011	r11
01100	r12
01101	r13
01110	r14
01111	r15

Code	Register
10000	r16
10001	r17
10010	r18
10011	r19
10100	r20
10101	r21
10110	r22
10111	r23
11000	r24
11001	r25
11010	r26
11011	r27
11100	r28
11101	r29
11110	r30
11111	r31

Exceptions

None

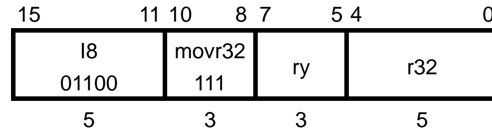
MOVE *ry*, *r32*

Move

Operation

$ry \Leftarrow r32$

Instruction Encoding



Description

The contents of general-purpose register *r32* is copied to general-purpose register *ry*, where *r32* is any of the 32 registers (r0 to r31) and *ry* is one of the eight registers visible to the 16-bit ISA.

To the 16-bit instructions, only eight of the 32 general-purpose registers are normally visible, r2 to r7, r16 and r17. Since the processor includes the full 32 registers of the 32-bit ISA mode, the 16-bit ISA includes the MOVE instructions to copy values between the eight 16-bit ISA registers and the remaining 24 registers of the full processor architecture. By virtue of the MOVE instructions, 16-bit routines can utilize all of the 32 general-purpose registers.

The encoding of the *r32* field in the 16-bit instruction code is as follows.

Code	Register
00000	r0
00001	r1
00010	r2
00011	r3
00100	r4
00101	r5
00110	r6
00111	r7
01000	r8
01001	r9
01010	r10
01011	r11
01100	r12
01101	r13
01110	r14
01111	r15

Code	Register
10000	r16
10001	r17
10010	r18
10011	r19
10100	r20
10101	r21
10110	r22
10111	r23
11000	r24
11001	r25
11010	r26
11011	r27
11100	r28
11101	r29
11110	r30
11111	r31

Exceptions

None

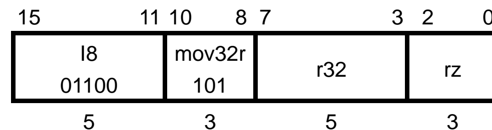
MOVE *r32*, *rz*

Move

Operation

r32 ⇐ *rz*

Instruction Encoding



Description

The contents of general-purpose register *rz* is copied to general-purpose register *r32*, where *rz* is one of the eight registers visible to the 16-bit ISA and *r32* is any of the 32 registers (r0 to r31).

To the 16-bit instructions, only eight of the 32 general-purpose registers are normally visible, r2 to r7, r16 and r17. Since the processor includes the full 32 registers of the 32-bit ISA mode, the 16-bit ISA includes the MOVE instructions to copy values between the eight 16-bit ISA registers and the remaining 24 registers of the full processor architecture. By virtue of the MOVE instructions, 16-bit routines can utilize all of the 32 general-purpose registers.

The encoding of the *r32* field in this 16-bit instruction code differs from that of the 32-bit ISA. The *r32* field, encoded as [2:0][4:3], denotes a general-purpose register as shown below.

Code	Register
00000	r0
00001	r8
00010	r16
00011	r24
00100	r1
00101	r9
00110	r17
00111	r25
01000	r2
01001	r10
01010	r18
01011	r26
01100	r3
01101	r11
01110	r19
01111	r27

Code	Register
10000	r4
10001	r12
10010	r20
10011	r28
10100	r5
10101	r13
10110	r21
10111	r29
11000	r6
11001	r14
11010	r22
11011	r30
11100	r7
11101	r15
11110	r23
11111	r31

Exceptions

None

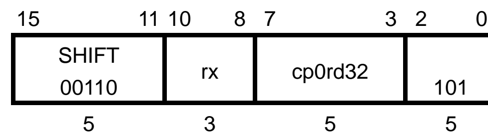
MTC0 *rx*, *cp0rd32*

Move to System Coprocessor 0 (CP0)

Operation

CP0 レジスタの *cp0rd32* ⇐ *rx*

Instruction Encoding



Description

The contents of general-purpose register *rx* is loaded into CP0 register *cp0rd32*.

In 16-bit ISA mode, this instruction can not access the Config1, Config2, Config3, IER and SSCR registers.

Once the MTC0 instruction writes to the Status, EPC or ErrorEPC register, at least two instructions must be executed before the ERET instruction. Otherwise, the operation is undefined.

Likewise, once the MTC0 instruction writes to the DEPC register, at least two instructions must be executed before the ERET instruction. Otherwise, the operation is undefined.

Because this instruction may alter the state of the virtual address translation system, the operation of load and store instructions immediately before and after this instruction is undefined.

Exceptions

Coprocessor Unusable exception

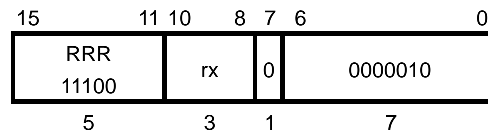
MTHI *rx*

Move To HI

Operation

HI ⇐ *rx*

Instruction Encoding



Description

The contents of general-purpose register *rx* is loaded into the HI register.

Exceptions

None

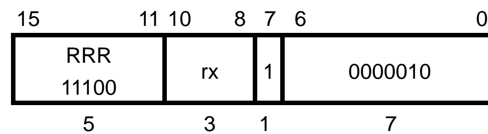
MTLO *rx*

Move To LO

Operation

LO ⇐ *rx*

Instruction Encoding



Description

The contents of general-purpose register *rx* is loaded into the LO register.

Exceptions

None

MULT *ry*, *rx*, *ry*

Multiply

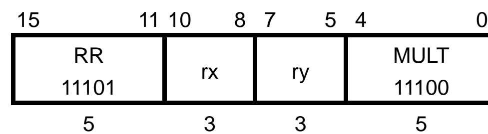
Operation

HI ⇐ high-order word of $(rx \times ry)$;

LO ⇐ low-order word of $(rx \times ry)$;

ry ⇐ low-order word of $(rx \times ry)$;

Instruction Encoding



Description

The contents of general-purpose register *rx* is multiplied by the contents of general-purpose register *ry*. Both *rx* and *ry* are treated as signed integers. The high-order word of the result is placed into the HI register, and the low-order word of the result is placed into the LO register and *ry*.

No Integer Overflow exception occurs under any circumstances.

Exceptions

None

Example

Assume that general-purpose registers r3 and r4 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MULT r4,r3,r4
```

evaluates:

(0x0123_4567 · 0x89AB_CDEF)
= 0xFF79_5E36_C94E_4629

Hence, the high-order word of the result, 0xFF79_5E36, is placed into the HI register, and the low-order word of the result, 0xC94E_4629, is placed into the LO and r4 registers.

MULT *rx*, *ry*

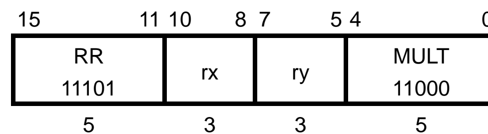
Multiply

Operation

HI ⇐ high-order word of ($rx \times ry$);

LO ⇐ high-order word of ($rx \times ry$);

Instruction Encoding



Description

The contents of general-purpose register *rx* is multiplied by the contents of general-purpose register *ry*. Both *rx* and *ry* are treated as signed integers. The high-order word of the result is placed into the HI register, and the low-order word of the result is placed into the LO register.

No Integer Overflow exception occurs under any circumstances.

Exceptions

None

Example

Assume that general-purpose registers r3 and r4 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MULT r3,r4
```

evaluates:

(0x0123_4567 · 0x89AB_CDEF)
= 0xFF79_5E36_C94E_4629

Hence, the high-order word of the result, 0xFF79_5E36, is placed into the HI register, and the low-order word of the result, 0xC94E_4629, is placed into the LO register.

MULTU *rx*, *ry*

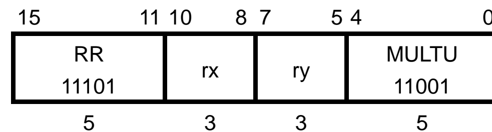
Multiply Unsigned

Operation

HI ⇐ high-order word of ($rx \times ry$);

LO ⇐ low-order word of ($rx \times ry$);

Instruction Encoding



Description

The contents of general-purpose register *rx* is multiplied by the contents of general-purpose register *ry*. Both *rx* and *ry* are treated as unsigned integers. The high-order word of the result is placed into the HI register, and the low-order word of the result is placed into the LO register.

No Integer Overflow exception occurs under any circumstances.

Exceptions

None

Example

Assume that general-purpose registers r3 and r4 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MULTU r3,r4
```

evaluates:

(0x0123_4567 · 0x89AB_CDEF)
= 0x009C_A39D_C94E_4629

Hence, the high-order word of the result, 0x009C_A39D, is placed into the HI register, and the low-order word of the result, 0xC94E_4629, is placed into the LO register.

MULTU *ry*, *rx*, *ry*

Multiply

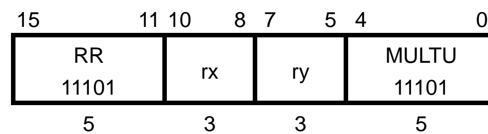
Operation

HI ⇐ high-order word of (rx × ry);

LO ⇐ low-order word of (rx × ry);

ry ⇐ low-order word of (rx × ry);

Instruction Encoding



Description

The contents of general-purpose register *rx* is multiplied by the contents of general-purpose register *ry*. Both *rx* and *ry* are treated as unsigned integers. The high-order word of the result is placed into the HI register, and the low-order word of the result is placed into the LO register and *ry*.

No Integer Overflow exception occurs under any circumstances.

Exceptions

None

Example

Assume that general-purpose registers r3 and r4 contain 0x0123_4567 and 0x89AB_CDEF respectively. Then, the instruction:

```
MULTU r4,r3,r4
```

evaluates:

(0x0123_4567 · 0x89AB_CDEF)
= 0x009C_A39D_C94E_4629

Hence, the high-order word of the result, 0x009C_A39D, is placed into the HI register, and the low-order word of the result, 0xC94E_4629, is placed into the LO and r4 registers.

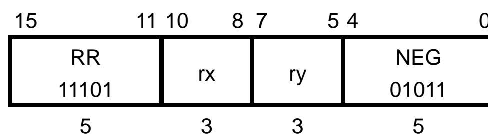
NEG *rx*, *ry*

Negate

Operation

$rx = 0 - ry$

Instruction Encoding



Description

This instruction performs 2's complement of the contents of general-purpose register *ry* and places the result into general-purpose register *rx*. It is implemented as the subtraction of *ry* from a value of zero.

Exceptions

None

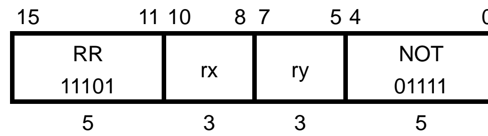
NOT *rx*, *ry*

NOT

Operation

$rx \Leftarrow ry$ NOR 0x0000_0000

Instruction Encoding



Description

This instruction performs 1's complement of the contents of general-purpose register *ry* and places the result into general-purpose register *rx*. Each bit in *ry* is inverted. It is implemented as the logical NOR of *ry* and a value of zero.

Exceptions

None

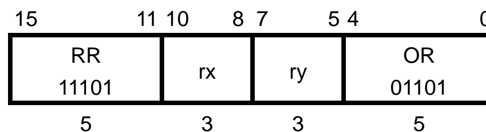
OR *rx*, *ry*

OR

Operation

$rx \Leftarrow rx \text{ OR } ry$

Instruction Encoding



Description

The contents of general-purpose register *rx* is ORed with the contents of general-purpose register *ry*, and the result is placed back into general-purpose register *rx*.

Exceptions

None

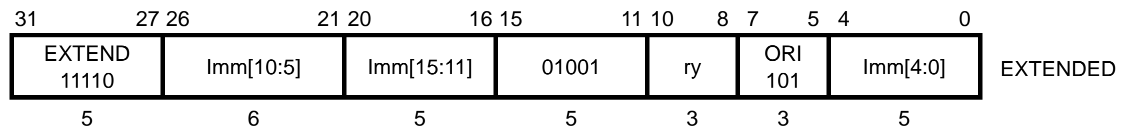
ORI *ry*, *immediate*

Logical OR Immediate

Operation

$ry \Leftarrow ry \text{ OR } (0^{16} \| \ immediate_{15..0})$

Instruction Encoding



Description

The 16-bit *immediate* is zero-extended and ORed with the contents of general-purpose register *ry*. The result is placed back into *ry*.

The *immediate* field is 16 bits in length. If the immediate size is larger than that, you need to put it in a general-purpose register and use the OR instruction (see 3.3.2, *32-Bit Constants*).

Exceptions

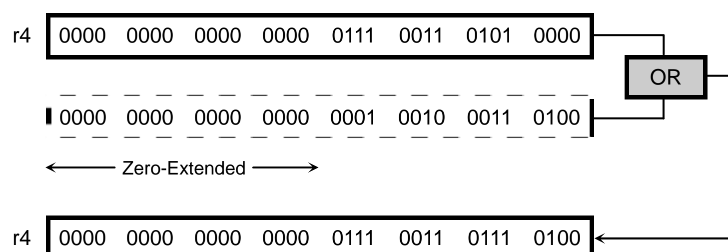
None

Example

Assume that register r4 contains 0x0000_7350. Then, the instruction:

```
ORI r4,0x1234
```

performs the logical OR between 0x0000_7350 and 0x0000_1234 and puts the result (0x0000_7374) back in r4, as shown below.



RESTORE *reg_list3*, *framesize4*

Restore Registers and Deallocate Stack Frame

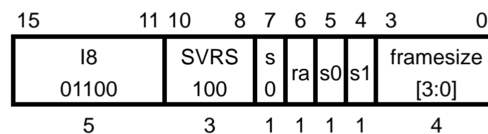
Operation

ra(r31) ⇐ Stack and/or s1(r17) ⇐ Stack and/or s0(r16) ⇐ Stack ;

if *framesize4* == 0 then sp(r29) ⇐ sp + 128 ;

else sp(r29) ⇐ sp + (0 || *framesize4* << 3) ;

Instruction Encoding



Description

The r31 (ra), r16 (s0) and/or r17 (s1) registers are restored from the memory stack if the corresponding ra, s0 and s1 bits of the instruction are set, and the stack pointer register (sp) is adjusted by the *framesize4* value. Higher numbered registers are loaded from higher stack addresses.

The encoding used for the *reg_list3* field is as follows:

reg_list3	ra	s0	s1
0x1	0	0	1
0x2	0	1	0
0x3	0	1	1
0x4	1	0	0
0x5	1	0	1
0x6	1	1	0
0x7	1	1	1

The *reg_list3* field must be non-zero; otherwise, the operation is unpredictable.

The 4-bit *framesize4* value is shifted left by three bits and zero-extended. A *framesize4* value of 0 is interpreted as a stack pointer adjustment of 128. Thus *framesize4* can be between +8 and +128 in increments of eight. If *framesize4* is outside this range, the instruction is EXTENDed, providing an 8-bit framesize field for stack pointer adjustment between 0 and +2040. The framesize field in the EXTENDed instruction is also shifted left by three bits.

If either of the two least-significant bits of the stack pointer is not zero, an Address Error exception occurs.

Operation Details

```
if framesize[3:0] = 0 then
    temp ⇐ sp (r29) + 128
else
    temp ⇐ sp (r29) + (0 || (framesize[3:0] << 3))
endif
temp2 ⇐ temp
if ra = 1 then
    temp ⇐ temp - 4
    r31   ⇐ Memory [temp]
endif
if s1 = 1 then
    temp ⇐ temp - 4
    r17   ⇐ Memory [temp]
endif
if s0 = 1 then
    temp ⇐ temp - 4
    r16   ⇐ Memory [temp]
endif
sp (r29) ⇐ temp2
```

Exceptions

Address Error exception

Programming Notes

The time required to execute this instruction varies, depending on the number of memory loads and the memory access time. In case of any interrupt during execution, the full sequence of operations will be restarted upon return from the interrupt.

RESTORE *reg_list3*, *xsregs*, *aregs*, *framesize8*

Restore Registers and Deallocate Stack Frame

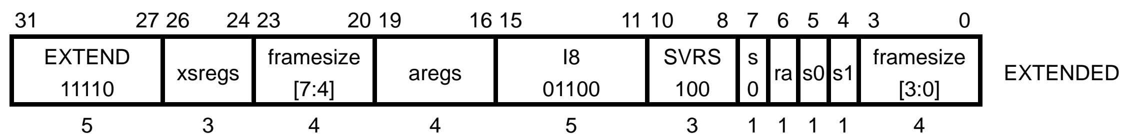
Operation

ra(r31) ⇐ Stack and/or [r18-r23, r30] ⇐ Stack and/or

s1(r17) ⇐ Stack and/or s0(r16) ⇐ Stack and/or [r4-r7] ⇐ Stack ;

sp(r29) ⇐ sp + (0 || *framesize8* << 3) ;

Instruction Encoding



Description

The r31 (ra) register is restored from the memory stack if the ra bit in the instruction is set. The r30 and r23-r18 registers are restored from the memory stack, as indicated by the value of the *xsregs* field. The r17 and/or r16 registers are restored from the memory stack if the corresponding s1 and s0 bits in the instruction are set. The r7 to r4 registers are restored from the memory stack, as indicated by the *aregs* field. The stack pointer register (sp) is adjusted by the *framesize8* value. Higher numbered registers are loaded from higher stack addresses.

For the interpretation of the *xsregs* field, see the *Operation Details* section. For the interpretation of the *aregs* field, see *the Interpretation of the aregs Field* section.

The encoding used for the *reg_list3* field is as follows:

reg_list3	ra	s0	s1
0x0	0	0	0
0x1	0	0	1
0x2	0	1	0
0x3	0	1	1
0x4	1	0	0
0x5	1	0	1
0x6	1	1	0
0x7	1	1	1

At least one register must be specified in any of the *reg_list3*, *xsregs* and *aregs* fields to be restored. If no register is specified, the behavior of the processor is unpredictable.

The 8-bit *framesize8* value is shifted left by three bits and zero-extended. Thus *framesize8* can be between 0 and +2040, in increments of eight.

If either of the two least-significant bits of the stack pointer is not zero, an Address Error exception occurs.

Interpretation of the aregs Field

In the standard MIPS ABIs (Application Binary Interfaces), registers r4-r7 are designated as a0-a3 for passing arguments to functions. When they are so used, they are saved on the stacks allocated not only by the caller of the routine being entered but also by its callee. In the standard MIPS ABIs, however, registers r4-r7 need not be restored on subroutine exit.

In other MIPS16e calling sequences, registers r4-r7 may be saved as static registers (i.e., registers preserved throughout the function) on the callee stack instead of the caller stack, and restored before return from the function.

The encoding used for the *aregs* field of the extended RESTORE instruction is the same as that used for the extended SAVE instruction, except that the RESTORE instruction ignores argument registers and handles only the registers treated as static.

The following table shows the encoding of the *aregs* field of the RESTORE instruction.

aregs Encoding (binary)	Registers Restored as Static Registers
0000	–
0001	r7
0010	r6, r7
0011	r5, r6, r7
1011	r4, r5, r6, r7
0100	–
0101	r7
0110	r6, r7
0111	r5, r6, r7
1000	–
1001	r7
1010	r6, r7
1100	–
1101	r7
1110	–
1111	Reserved

Operation Details

```
temp   ⇐ sp (r29) + (0 || (framesize[7:0] << 3))
temp2 ⇐ temp
if ra = 1 then
   temp ⇐ temp - 4
   r31   ⇐ Memory [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
                                 r30   ⇐ Memory [temp]
                            endif
                            temp ⇐ temp - 4
                            r23   ⇐ Memory [temp]
                       endif
                       temp ⇐ temp - 4
                       r22   ⇐ Memory [temp]
                  endif
                  temp ⇐ temp - 4
                  r21   ⇐ Memory [temp]
             endif
             temp ⇐ temp - 4
             r20   ⇐ Memory [temp]
        endif
        temp ⇐ temp - 4
        r19   ⇐ Memory [temp]
   endif
   temp ⇐ temp - 4
   r18   ⇐ Memory [temp]
endif
if s1 = 1 then
   temp ⇐ temp - 4
   r17   ⇐ Memory [temp]
endif
if s0 = 1 then
   temp ⇐ temp - 4
   r16   ⇐ Memory [temp]
endif
case aregs of (in binary)
   0000, 0100, 1000, 1100, 1110 : astatic ⇐ 0
   0001, 0101, 1001, 1101 : astatic ⇐ 1
   0010, 0110, 1010 : astatic ⇐ 2
   0011, 0111 : astatic ⇐ 3
```

```
        1011 : astatic ⇐ 4
        otherwise : UNPREDICTABLE
    endcase
    if astatic > 0 then
        temp ⇐ temp - 4
        r7    ⇐ Memory [temp]
        if astatic > 1 then
            temp ⇐ temp - 4
            r6    ⇐ Memory [temp]
            if astatic > 2 then
                temp ⇐ temp - 4
                r5    ⇐ Memory [temp]
                if astatic > 3 then
                    temp ⇐ temp - 4
                    r4    ⇐ Memory [temp]
                endif
            endif
        endif
    endif
    sp (r29) ⇐ temp2
```

Exceptions

Address Error exception

Programming Notes

The time required to execute this instruction varies, depending on the number of memory loads and the memory access time. In case of any interrupt during execution, the full sequence of operations will be restarted upon return from the interrupt.

The behavior of the processor is unpredictable if a reserved value of 1111 is given in the *aregs* field.

SADD *ry*, *rx*, *ry*

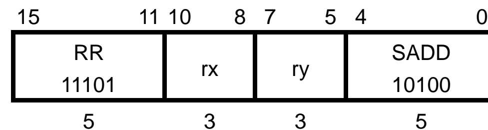
Saturated Add

Operation

if overflow on $rx + ry$ then $ry \Leftarrow$ 0x7FFF_FFFF ($rx \geqq 0$) or 0x8000_0000 ($rx < 0$)

else $ry \Leftarrow rx + ry$

Instruction Encoding



Description

The contents of general-purpose register *rx* is added to the contents of general-purpose register *ry*. The sum saturates to the largest representable positive number (0x7FFF_FFFF) on overflow and to the smallest representable negative number (0x8000_0000) on underflow. The result is placed into *ry*. If neither overflow nor underflow occurs, the sum of *rx* and *ry* is placed into *ry*. Both *rx* and *ry* are treated as signed integers.

An Integer Overflow exception never occurs on overflow.

Exceptions

None

SAVE *reg_list3, framesize4*

Save Registers and Set up Stack Frame

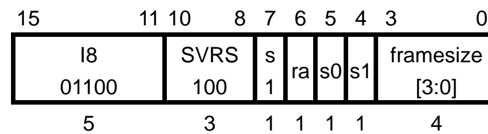
Operation

Stack ⇐ ra(r31) and/or Stack ⇐ s1(r17) and/or Stack ⇐ s0(r16) ;

if *framesize4* == 0 then sp(r29) ⇐ sp - 128 ;

else sp(r29) ⇐ sp - (0 || *framesize4* << 3) ;

Instruction Encoding



Description

The r31 (ra), r16 (s0) and/or r17 (s1) registers are saved to the memory stack if the corresponding ra, s0 and s1 bits of the instruction are set, and the stack pointer register (sp) is adjusted by the *framesize4* value. Higher numbered registers are stored to higher stack addresses.

The encoding used for the *reg_list3* field is as follows:

reg_list3	ra	s0	s1
0x1	0	0	1
0x2	0	1	0
0x3	0	1	1
0x4	1	0	0
0x5	1	0	1
0x6	1	1	0
0x7	1	1	1

The *reg_list3* field must be non-zero; otherwise, the operation is unpredictable.

The 4-bit *framesize4* value is shifted left by three bits and zero-extended. A *framesize4* value of 0 is interpreted as a stack pointer adjustment of 128. Thus *framesize4* can be between +8 and +128 in increments of eight. If *framesize4* is outside this range, the instruction is EXTENDed, providing an 8-bit framesize field for stack pointer adjustment between 0 and +2040. The framesize field in the EXTENDed instruction is also shifted left by three bits.

If either of the two least-significant bits of the stack pointer is not zero, an Address Error exception occurs.

Operation Details

```
temp ⇐ sp (r29)
if ra = 1 then
    temp ⇐ temp - 4
    Memory [temp] ⇐ r31
endif
if s1 = 1 then
    temp ⇐ temp - 4
    Memory [temp] ⇐ r17
endif
if s0 = 1 then
    temp ⇐ temp - 4
    Memory [temp] ⇐ r16
endif
if framesize[3:0] = 0 then
    temp ⇐ sp (r29) - 128
else
    temp ⇐ sp (r29) - (0 || (framesize[3:0] << 3))
endif
sp (r29) ⇐ temp
```

Exceptions

Address Error exception

Programming Notes

The time required to execute this instruction varies, depending on the number of memory loads and the memory access time. In case of any interrupt during execution, the full sequence of operations will be restarted upon return from the interrupt.

SAVE *reg_list3*, *xsregs*, *aregs*, *framesize8*

Save Registers and Set up Stack Frame

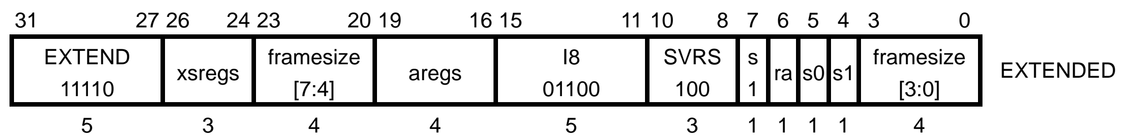
Operation

Stack ⇐ ra(r31) and/or Stack ⇐ [r18-r23, r30] and/or

Stack ⇐ s1(r17) and/or Stack ⇐ s0(r16) and/or Stack ⇐ [r4-r7] ;

sp(r29) ⇐ sp - (0 || *framesize8* << 3) ;

Instruction Encoding



Description

Registers r4-r7 are saved on the memory stack as arguments, as indicated by the value of the *aregs* field. Register r31 (ra) is saved on the memory stack if the corresponding ra bit in the instruction is set. Registers r18-r23 and r30 are saved on the memory stack, as indicated by the *xsregs* field. Registers r16 (s0) and/or r17 (s1) are saved on the memory stack if the corresponding s0 and s1 bits in the instruction are set. Registers r4-r7 are saved on the stack as static registers as indicated by the *aregs* field. The stack pointer register (sp) is adjusted by the *framesize8* value. Higher numbered registers are loaded from higher stack addresses.

For the interpretation of the *xsregs* field, see the *Operation Details* section. For the interpretation of the *aregs* field, see *the Interpretation of the aregs Field* section.

The encoding used for the *reg_list3* field is as follows:

reg_list3	ra	s0	s1
0x0	0	0	0
0x1	0	0	1
0x2	0	1	0
0x3	0	1	1
0x4	1	0	0
0x5	1	0	1
0x6	1	1	0
0x7	1	1	1

At least one register must be specified in any of the *reg_list3*, *xsregs* and *aregs* fields to be saved. If no register is specified, the behavior of the processor is unpredictable.

The 8-bit *framesize8* value is shifted left by three bits and zero-extended. Thus *framesize8* can be between 0 and +2040, in increments of eight.

If either of the two least-significant bits of the stack pointer is not zero, an Address Error exception occurs.

Interpretation of the aregs Field

In the standard MIPS ABIs (Application Binary Interfaces), registers r4-r7 are designated as a0-a3 for passing arguments to functions. When they are so used, they are saved on the stacks allocated not only by the caller of the routine being entered but also by its callee.

In other MIPS16e calling sequences, registers r4-r7 may be saved as static registers (i.e., registers preserved throughout the function) on the callee stack instead of the caller stack.

The encoding of the *aregs* field allows for zero to four argument registers, zero to four static registers and mixtures of the two. Registers are bound to arguments (a0, a1, a2 and a3) in ascending order, and thus assigned to static values (r7, r6, r5 and r4) in the reverse order.

The following table shows the encoding of the *aregs* field of the SAVE instruction.

aregs Encoding (binary)	Registers Saved as Argument Registers	Registers Saved as Static Registers
0000	–	–
0001	–	r7
0010	–	r6, r7
0011	–	r5, r6, r7
1011	–	r4, r5, r6, r7
0100	a0(r4)	–
0101	a0(r4)	r7
0110	a0(r4)	r6, r7
0111	a0(r4)	r5, r6, r7
1000	a0(r4), a1(r5)	–
1001	a0(r4), a1(r5)	r7
1010	a0(r4), a1(r5)	r6, r7
1100	a0(r4), a1(r5), a2(r6)	–
1101	a0(r4), a1(r5), a2(r6)	r7
1110	a0(r4), a1(r5), a2(r6), a3(r7)	–
1111	Reserved	Reserved

Operation Details

```
temp ⇐ sp (r29)
case aregs of (in binary)
    0000, 0001, 0010, 0011, 1011 : args ⇐ 0
    0100, 0101, 0110, 0111 : args ⇐ 1
    1000, 1001, 1010 : args ⇐ 2
    1100, 1101 : args ⇐ 3
    1110 : args ⇐ 4
    otherwise : UNPREDICTABLE
endcase
if args > 0 then
    Memory [temp] ⇐ r4
    if args > 1 then
        Memory [temp + 4] ⇐ r5
        if args > 2 then
            Memory [temp + 8] ⇐ r6
            if args > 3 then
                Memory [temp + 12] ⇐ r7
            endif
        endif
    endif
endif
if ra = 1 then
    temp ⇐ temp - 4
    Memory [temp] ⇐ r31
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
                            Memory [temp] ⇐ r30
                        endif
                        temp ⇐ temp - 4
                        Memory [temp] ⇐ r23
                    endif
                    temp ⇐ temp - 4
                    Memory [temp] ⇐ r22
                endif
                temp ⇐ temp - 4
                Memory [temp] ⇐ r21
            endif
            temp ⇐ temp - 4
            Memory [temp] ⇐ r20
        endif
        temp ⇐ temp - 4
        Memory [temp] ⇐ r19
```

```
        endif
        temp ⇐ temp - 4
        Memory [temp] ⇐ r18
    endif
    if s1 = 1 then
        temp ⇐ temp - 4
        Memory [temp] ⇐ r17
    endif
    if s0 = 1 then
        temp ⇐ temp - 4
        Memory [temp] ⇐ r16
    endif
    case aregs of (in binary)
        0000, 0100, 1000, 1100, 1110 : astatic ⇐ 0
        0001, 0101, 1001, 1101 : astatic ⇐ 1
        0010, 0110, 1010 : astatic ⇐ 2
        0011, 0111 : astatic ⇐ 3
        1011 : astatic ⇐ 4
        otherwise : UNPREDICTABLE
    endcase

    if astatic > 0 then
        temp ⇐ temp - 4
        Memory [temp] ⇐ r7
        if astatic > 1 then
            temp ⇐ temp - 4
            Memory [temp] ⇐ r6
            if astatic > 2 then
                temp ⇐ temp - 4
                Memory [temp] ⇐ r5
                if astatic > 3 then
                    temp ⇐ temp - 4
                    Memory [temp] ⇐ r4
                endif
            endif
        endif
    endif
    temp ⇐ sp (r29) - (0 || (framesize[7:0] << 3))
    sp (r29) ⇐ temp
```

Exceptions

Address Error exception

Programming Notes

The time required to execute this instruction varies, depending on the number of memory loads and the memory access time. In case of any interrupt during execution, the full sequence of operations will be restarted upon return from the interrupt.

The behavior of the processor is unpredictable if a reserved value of 1111 is given in the *aregs* field.

SB *ry*, *offset* (*base*)

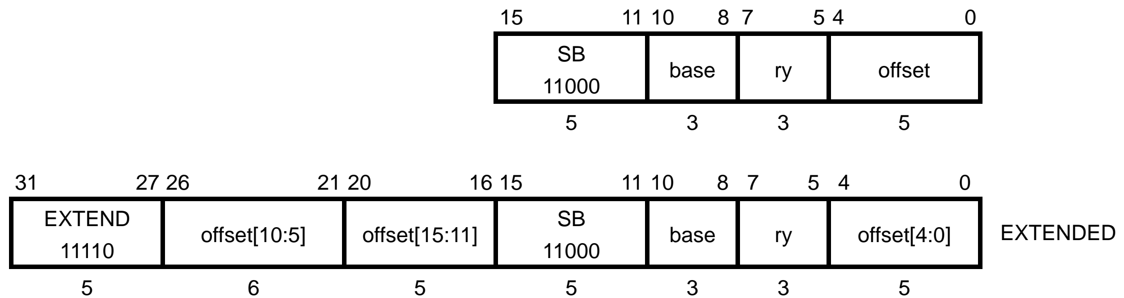
Store Byte

Operation

ry = {zero-extend (*offset*) + (*base*)}

(EXTENDED) *ry* = {sign-extend (*offset*) + (*base*)}

Instruction Encoding



Description

The 5-bit immediate *offset* is zero-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The least-significant byte in general-purpose register *ry* is stored at the memory location addressed by the EA.

The three high-order bytes in *ry* is simply ignored; so there is no distinction between signed and unsigned stores.

With the 5-bit *offset* field, the offset range is 0 to 31. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

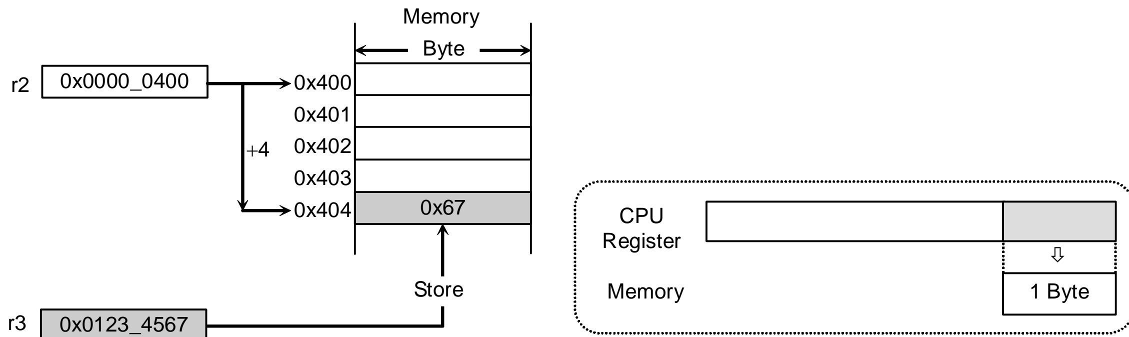
Address Error exception

Example

Assume that registers r2 and r3 contain 0x0000_0400 and 0x0123_4567 respectively. Then, executing the instruction:

```
SB r3,4(r2)
```

stores 0x67 to the memory location at address 0x404.



SB *ry*, *offset* (fp)

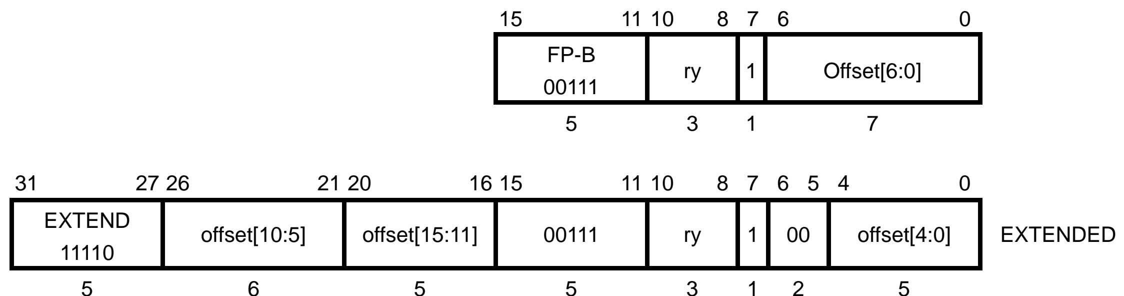
Store Byte

Operation

ry = {zero-extend (*offset*) + (fp)}

(EXTENDED) *ry* = {sign-extend (*offset*) + (fp)}

Instruction Encoding



Description

The 7-bit immediate *offset* is zero-extended and added to the contents of the fp register (r30) to form an effective address (EA). The least-significant byte in general-purpose register *ry* is stored at the memory location addressed by the EA.

The three high-order bytes in *ry* are simply ignored; so there is no distinction between signed and unsigned stores.

With the 7-bit *offset* field, the offset range is 0 to 127. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

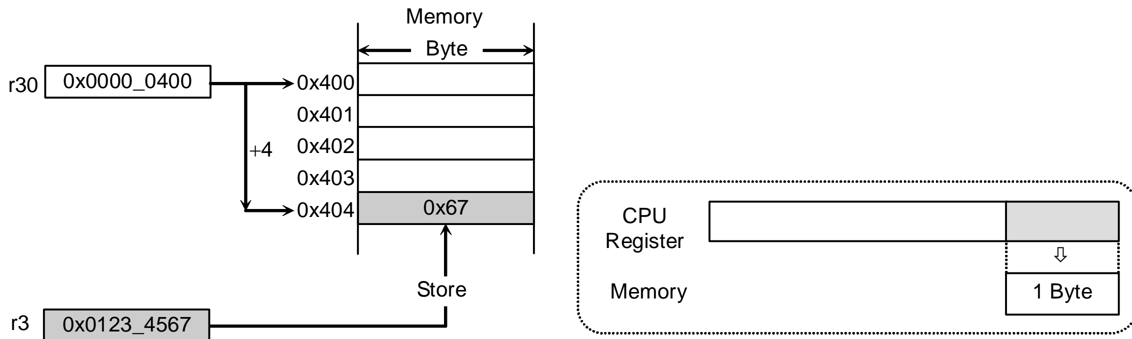
Address Error exception

Example

Assume that the fp (r30) and r3 registers contain 0x0000_0400 and 0x0123_4567 respectively. Then, executing the instruction:

```
B r3,4(fp)
```

stores 0x67 to the memory location at address 0x0404.



SB *ry*, *offset* (sp)

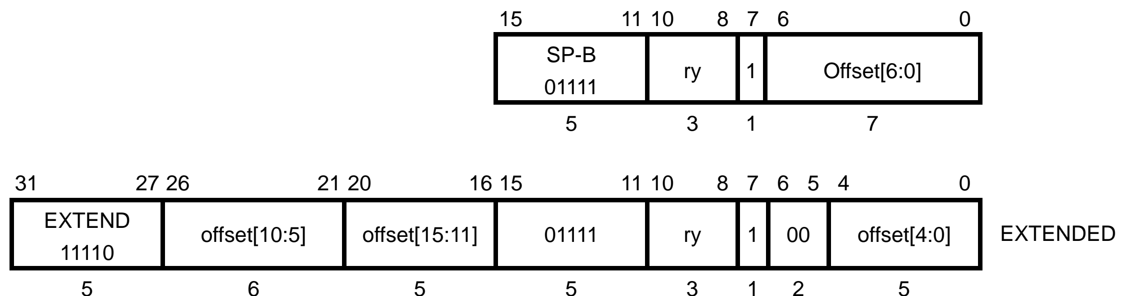
Store Byte

Operation

ry = {zero-extend (*offset*) + (sp)}

(EXTENDED) *ry* = {sign-extend (*offset*) + (sp)}

Instruction Encoding



Description

The 7-bit immediate *offset* is zero-extended and added to the contents of the sp register (r29) to form an effective address (EA). The least-significant byte in general-purpose register *ry* is stored at the memory location addressed by the EA.

The three high-order bytes in *ry* are simply ignored; so there is no distinction between signed and unsigned stores.

With the 7-bit *offset* field, the offset range is 0 to 127. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

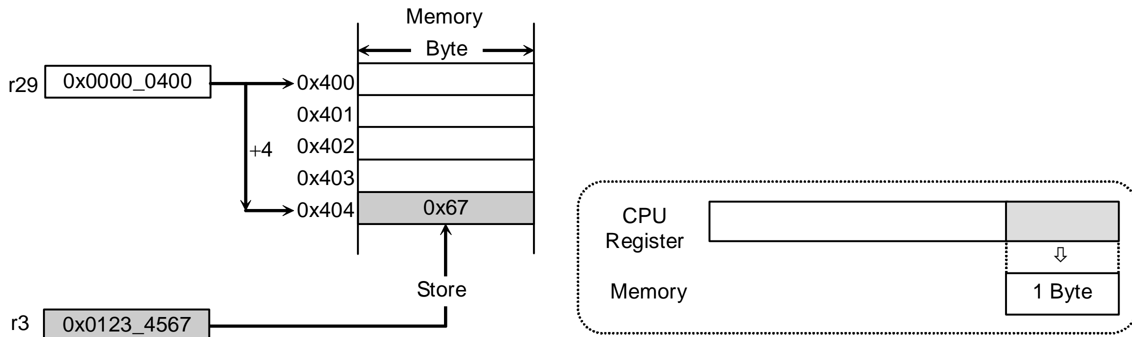
Address Error exception

Example

Assume that the sp (r29) and r3 registers contain 0x0000_0400 and 0x0123_4567 respectively. Then, executing the instruction:

```
SB r3,4(sp)
```

stores 0x67 to the memory location at address 0x404.



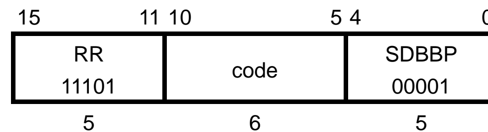
SDBBP *code*

Software Debug Breakpoint

Operation

Software debug breakpoint exception

Instruction Encoding



Description

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

The *code* field in the SDBBP instruction is available for use as software parameters to pass additional information. The exception handler can retrieve it by loading the contents of the memory word containing the instruction. See Section 9.3, *Debug Exceptions*, for details.

The SDBBP instruction may not be used within the user's program; it is intended for use by development tools. Executing the SDBBP instruction on a device without EJTAG causes a Reserved Instruction exception.

Exceptions

Debug Breakpoint exception
Reserved Instruction exception

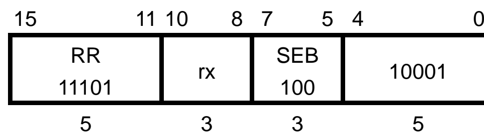
SEB *rx*

Sign-Extend Byte

Operation

$rx \Leftarrow (rx[7])^{24} \parallel rx[7:0]$

Instruction Encoding



Description

The least-significant byte in general-purpose register *rx* is sign-extended. The result is placed back into *rx*.

Exceptions

None

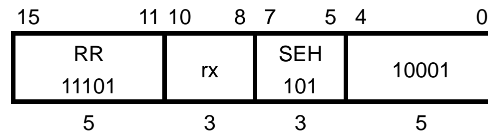
SEH *rx*

Sign-Extend Halfword

Operation

$rx \Leftarrow (rx[15])^{16} \ || \ rx[15{:}0];$

Instruction Encoding



Description

The low-order halfword in general-purpose register *rx* is sign-extended. The result is placed back into *rx*.

Exceptions

None

SH *ry*, *offset* (*base*)

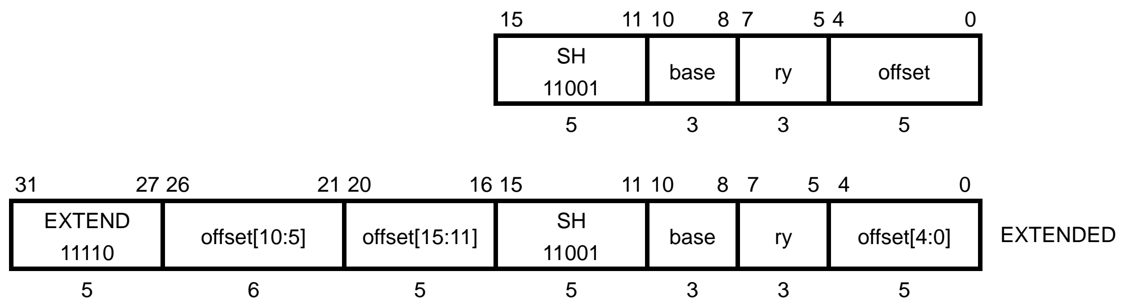
Store Halfword

Operation

ry = {zero-extend (*offset* || 0) + (*base*)}

(EXTENDED) *ry* = {sign-extend (*offset*) + (*base*)}

Instruction Encoding



Description

The 5-bit immediate *offset* is shifted left by one bit, zero-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The low-order halfword in general-purpose register *ry* is stored at the memory location addressed by the EA.
The high-order halfword in *ry* is simply ignored; so there is no distinction between signed and unsigned stores.

Since the 5-bit *offset* is shifted left by one bit, the offset range is 0 to 62, in increments of two. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767. When EXTENDed, the *offset* operand is not shifted at all.

Exceptions

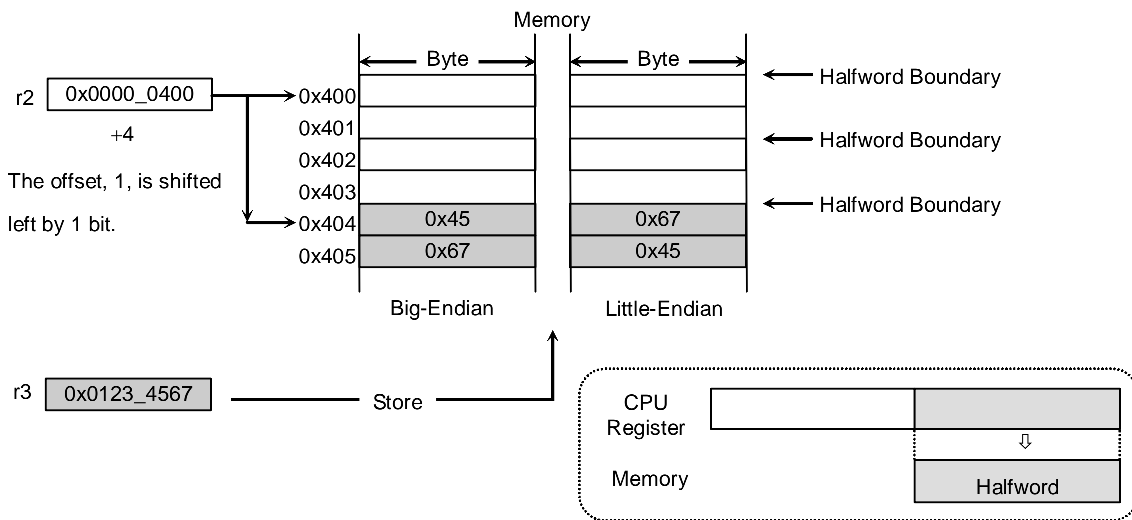
Address Error exception

Example

```
SH r3,4(r2)
```

Assume that registers r2 and r3 contain 0x0000_0400 and 0x0123_4567 respectively. Since the offset value is shifted left by one bit by the processor hardware, the assembler/linker turns the specified offset (4 or binary 0100) into a code of 2 (binary 0010). Thus the instruction code for this store instruction becomes 0xCA62.

In big-endian mode, 0x45 and 0x67 are stored to the memory locations at addresses 0x404 and 0x405 respectively. In little-endian mode, 0x67 and 0x45 are stored to the memory locations at addresses 0x404 and 0x405 respectively.



SH *ry*, *offset* (fp)

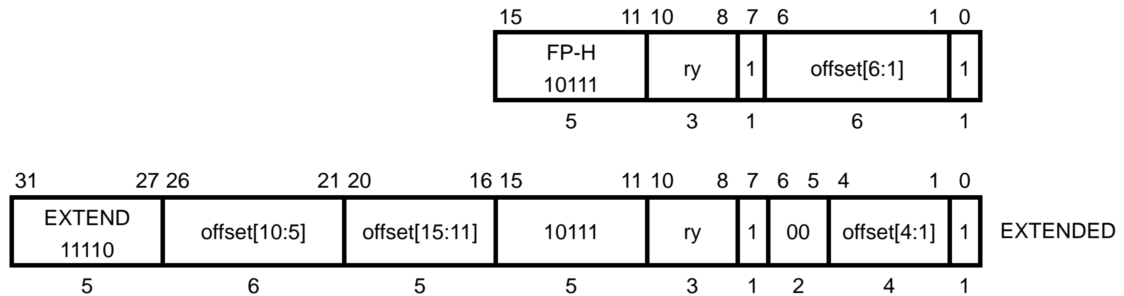
Store Halfword

Operation

ry = {zero-extend ($offset$ || 0) + (fp)}

(EXTENDED) ry = {sign-extend ($offset$ || 0) + (fp)}

Instruction Encoding



Description

The 6-bit immediate *offset* is shifted left by one bit, zero-extended and added to the contents of the fp register (r30) to form an effective address (EA). The low-order halfword in general-purpose register *ry* is stored at the memory location addressed by the EA.

The high-order halfword in *ry* is simply ignored; so there is no distinction between signed and unsigned stores.

Since the 6-bit *offset* is shifted left by one bit, the offset range is 0 to 126, in increments of two. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate. When EXTENDed, the *offset* operand is shifted left by one bit to allow an offset of -32768 to +32766.

Exceptions

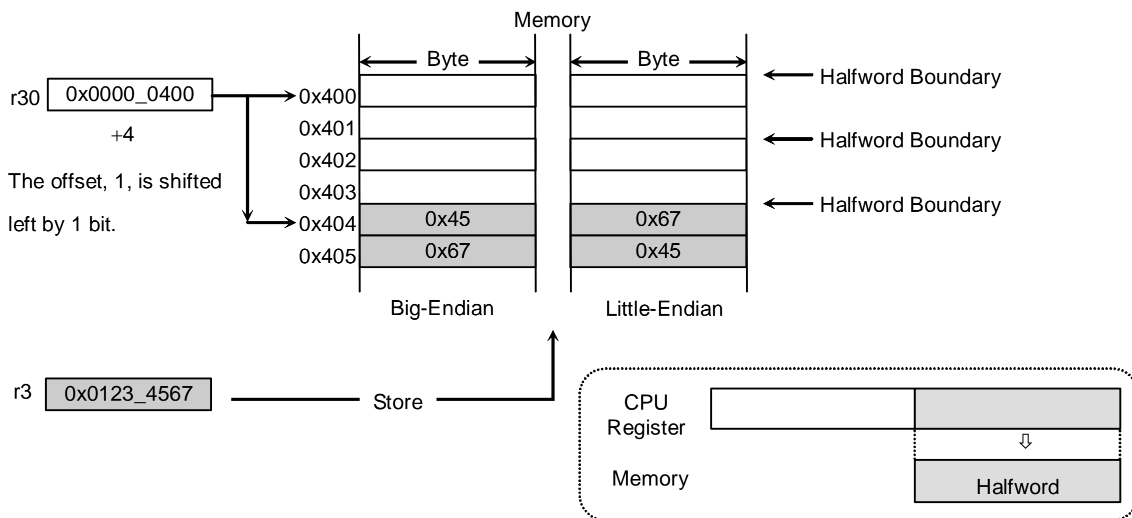
Address Error exception

Example

```
SH r3,4(fp)
```

Assume that fp (r30) and r3 registers contain 0x0000_0400 and 0x0123_4567 respectively. Since the offset value is shifted left by one bit by the processor hardware, the assembler/linker turns the specified offset (4 or binary 0100) into a code of 2 (binary 0010). Thus the instruction code for this store instruction becomes 0xBB85.

In big-endian mode, 0x45 and 0x67 are stored to the memory locations at addresses 0x404 and 0x405 respectively. In little-endian mode, 0x67 and 0x45 are stored to the memory locations at addresses 0x404 and 0x405 respectively.



SH *ry*, *offset* (sp)

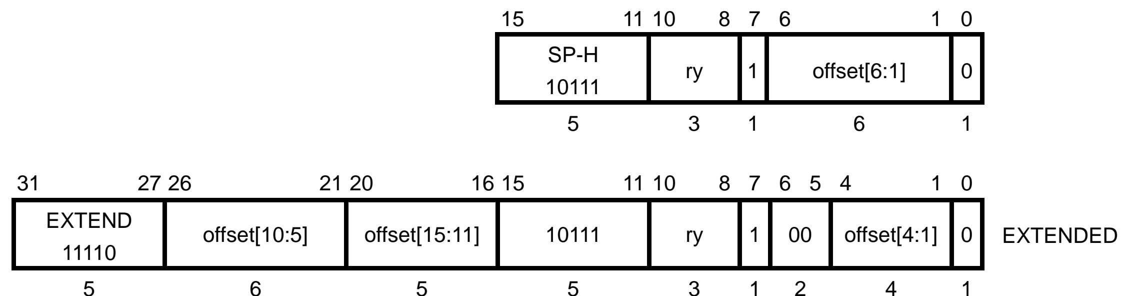
Store Halfword

Operation

ry = {zero-extend (*offset* || 0) + (sp)}

(EXTENDED) *ry* = {sign-extend (*offset* || 0) + (sp)}

Instruction Encoding



Description

The 6-bit immediate *offset* is shifted left by one bit, zero-extended and added to the contents of the sp (r29) register to form an effective address (EA). The low-order halfword in general-purpose register *ry* is stored at the memory location addressed by the EA.

The high-order halfword in *ry* is simply ignored; so there is no distinction between signed and unsigned stores.

Since the 6-bit *offset* is shifted left by one bit, the offset range is 0 to 126, in increments of two. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate. When EXTENDed, the *offset* operand is shifted left by one bit allow an offset of -32768 to +32766.

Exceptions

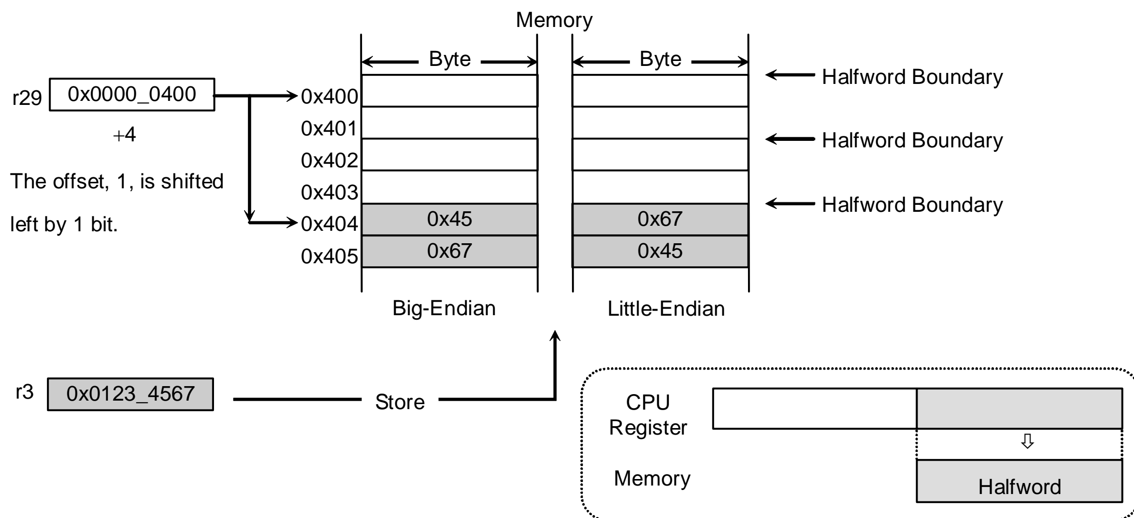
Address Error exception

Example

```
SH r3,4(sp)
```

Assume that the sp (r29) and r3 registers contain 0x0000_0400 and 0x0123_4567 respectively. Since the offset value is shifted left by one bit by the processor hardware, the assembler/linker turns the specified offset (4 or binary 0100) into a code of 2 (binary 0010). Thus the instruction code for this store instruction becomes 0xBB84.

In big-endian mode, 0x45 and 0x67 are stored to the memory locations at addresses 0x404 and 0x405 respectively. In little-endian mode, 0x67 and 0x45 are stored to the memory locations at addresses 0x404 and 0x405 respectively.



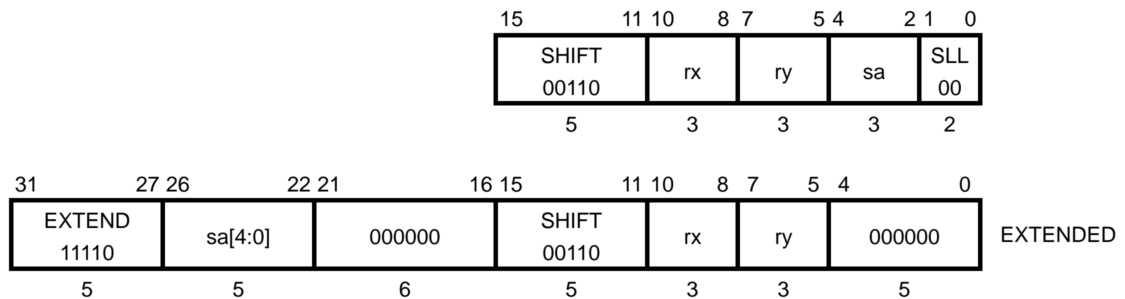
SLL *rx*, *ry*, *sa*

Shift Left Logical

Operation

$rx \Leftarrow ry << sa$

Instruction Encoding



Description

The contents of general-purpose register *ry* is shifted left by *sa* bits. Zeros are supplied to the vacated positions on the right. The result is placed into general-purpose register *rx*. The *sa* field is only 3-bits wide. Thus the shift amount is restricted to 1 to 8. The *sa* value of 000 is defined as a shift of 8 bits.

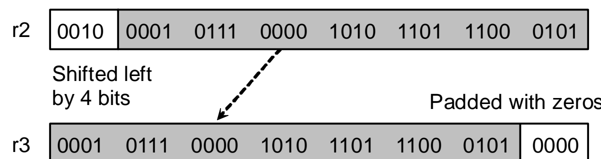
If the shift amount does not fit in the *sa* field, the instruction is EXTENDed to provide a full 5-bit field for a shift of 0 to 31.

Example

Assume that register r2 contains 0x2170_ADC5. Then, executing the instruction:

```
SLL r3,r2,4
```

places 0x170A_DC50 in register r3, as shown below.



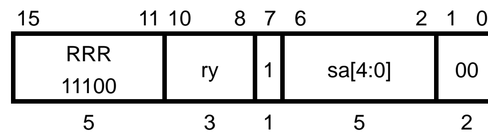
SLL *ry*, *sa5*

Shift Left Logical

Operation

$ry \Leftarrow ry << sa5$

Instruction Encoding



Description

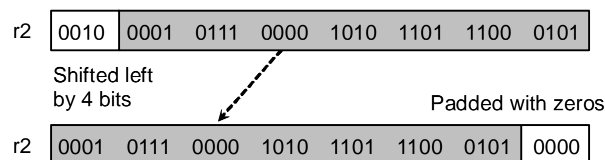
The contents of general-purpose register *ry* is shifted left by *sa* bits. Zeros are supplied to the vacated positions on the right. The result is placed back into *ry*. The *sa* field is 5-bits wide; thus the possible shift amount is 1 to 31. The *sa* value may not be 00000.

Example

Assume that register r2 contains 0x2170_ADC5. Then, executing the instruction:

```
SLL r2,4
```

places 0x170A_DC50 in register r2, as shown below.



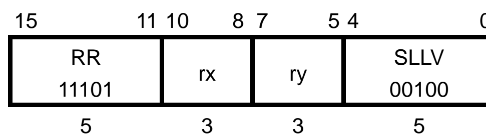
SLLV *ry*, *rx*

Shift Left Logical Variable

Operation

ry << 5 LSBs of *rx*

Instruction Encoding



Description

The contents of general-purpose register *ry* is shifted left the number of bits specified by the five least-significant bits of general-purpose register *rx*. Zeros are supplied to the vacated positions on the right. The result is placed back into general-purpose register *ry*.

Exceptions

None

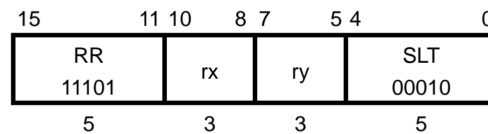
SLT *rx*, *ry*

Set On Less Than

Operation

if *rx* < *ry* then t8 ⇐ 1; else t8 ⇐ 0

Instruction Encoding



Description

The contents of general-purpose register *rx* is compared to the contents of general-purpose register *ry*. Both *rx* and *ry* are treated as signed integers. If *rx* is less than *ry*, condition code register t8 (r24) is set to one. Otherwise, t8 is set to zero.

No Integer Overflow exception occurs under any circumstances. The comparison is valid even if the subtraction performed for comparison results in overflow.

Exceptions

None

SLTI *rx*, *immediate*

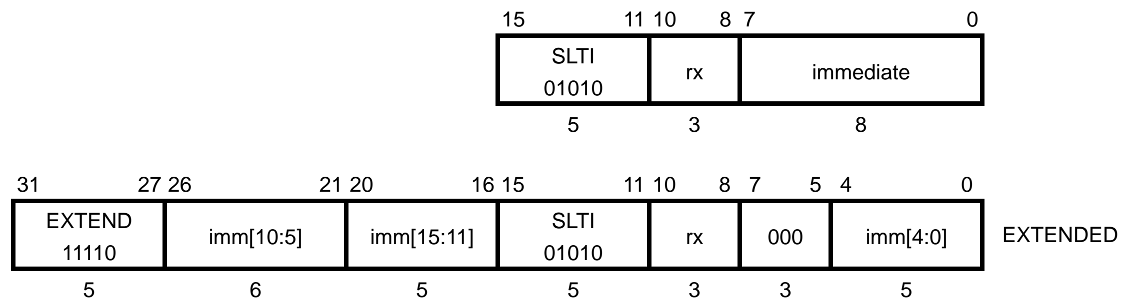
Set On Less Than Immediate

Operation

if $rx < 0^{24} \parallel (immediate_{7..0})$ then t8 ⇐ 1; else t8 ⇐ 0

(EXTENDED) if $rx < (immediate_{15})^{16} \parallel (immediate_{15..0})$ then t8 ⇐ 1; else t8 ⇐ 0

Instruction Encoding



Description

The 8-bit *immediate* is *zero*-extended and compared to the contents of general-purpose register *rx*. The *immediate* and *rx* are compared as *signed* integers. If *rx* is less than *immediate*, condition code register t8 (r24) is set to 1. Otherwise, t8 is set to zero.

No Integer Overflow exception occurs under any circumstances. The comparison is valid even if the subtraction performed for comparison results in overflow.

With the 8-bit *immediate* field, the immediate range is 0 to 255. If a number is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

None

SLTIU *rx*, *immediate*

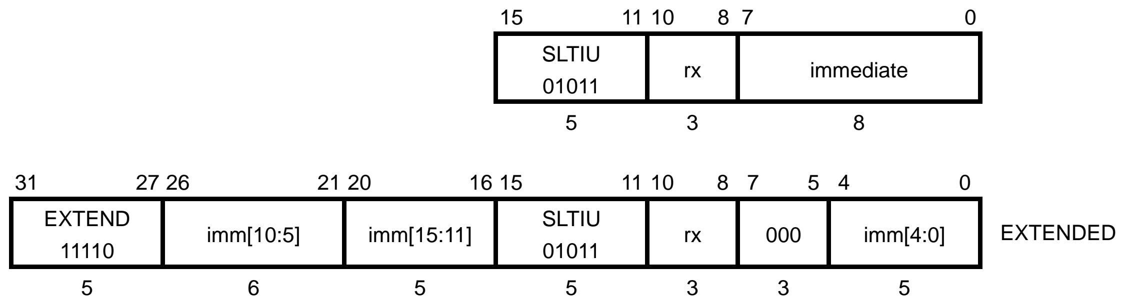
Set On Less Than Immediate Unsigned

Operation

$\text{if } (0 \parallel rx) < 0^{25} \parallel (immediate_{7..0}) \text{ then t8} \Leftarrow 1\text{; else t8} \Leftarrow 0$

(EXTENDED) $\text{if } (0 \parallel rx) < (immediate_{15})^{17} \parallel (immediate_{15..0}) \text{ then t8} \Leftarrow 1\text{; else t8} \Leftarrow 0$

Instruction Encoding



Description

The 8-bit *immediate* is zero-extended and compared to the contents of general-purpose register *rx*. The *immediate* and *rx* are compared as unsigned integers. If *rx* is less than *immediate*, condition code register t8 (r24) is set to one. Otherwise, t8 is set to zero.

No Integer Overflow exception occurs under any circumstances. The comparison is valid even if the subtraction performed for comparison results in overflow.

With the 8-bit *immediate* field, the immediate range is 0 to 255. If a number is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

None

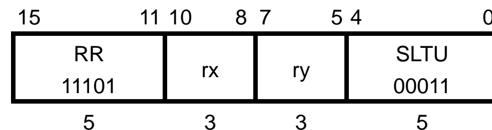
SLTU *rx*, *ry*

Set On Less Than Unsigned

Operation

if (0 || *rx*) < (0 || *ry*) then t8 ⇐ 1; else t8 ⇐ 0

Instruction Encoding



Description

The contents of general-purpose register *rx* is compared to the contents of general-purpose register *ry*. Both *rx* and *ry* are treated as unsigned integers. If *rx* is less than *ry*, condition code register t8 (r24) is set to one. Otherwise, t8 is set to zero.

No Integer Overflow exception occurs under any circumstances. The comparison is valid even if the subtraction performed for comparison results in overflow.

Exceptions

None

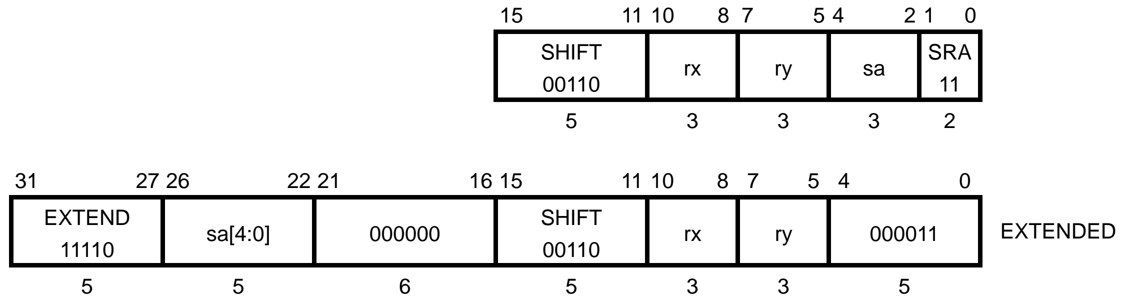
SRA *rx*, *ry*, *sa*

Shift Right Arithmetic

Operation

rx ⇐ *ry* >> *sa*

Instruction Encoding



Description

The contents of general-purpose register *ry* is shifted right by *sa* bits. The sign bit is copied to the vacated positions on the left. The result is placed into general-purpose register *rx*. The *sa* field is only 3-bits wide. Thus the shift amount is restricted to 1 to 8. The *sa* value of 000 is defined as a shift of 8 bits.

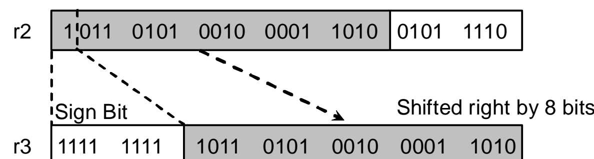
If the shift amount does not fit in the *sa* field, the instruction is EXTENDed to provide a full 5-bit field for shift of 0 to 31.

Example

Assume that register r2 contains 0xB521_AE5E. Then, executing the instruction:

```
SRA r3,r2,8
```

places 0xFFB5_21AE in register r3, as shown below.



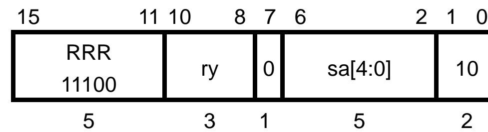
SRA *ry*, *sa5*

Shift Right Arithmetic

Operation

$ry \Leftarrow ry >> sa5$

Instruction Encoding



Description

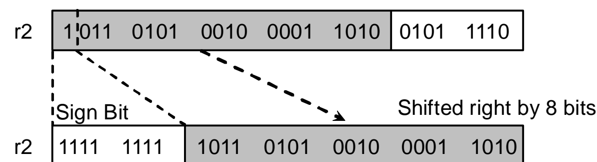
The contents of general-purpose register *ry* is shifted right by *sa* bits. The sign bit is copied to the vacated positions on the left. The result is placed back into *ry*. The *sa* field is 5-bits wide; thus the possible shift amount is 1 to 31. The *sa* value may not be 00000.

Example

Assume that register r2 contains 0xB521_AE5E. Then, executing the instruction:

```
SRA r2,8
```

places 0xFFB5_21AE back in register r2, as shown below.



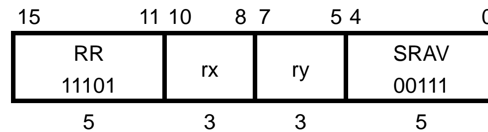
SRAV *ry*, *rx*

Shift Right Arithmetic Variable

Operation

ry >> 5 LSBs of *rx*

Instruction Encoding



Description

The contents of general-purpose register *ry* is shifted right the number of bits specified by the five least-significant bits of general-purpose register *rx*. The sign bit is copied to the vacated positions on the left. The result is placed back into general-purpose register *ry*.

Exceptions

None

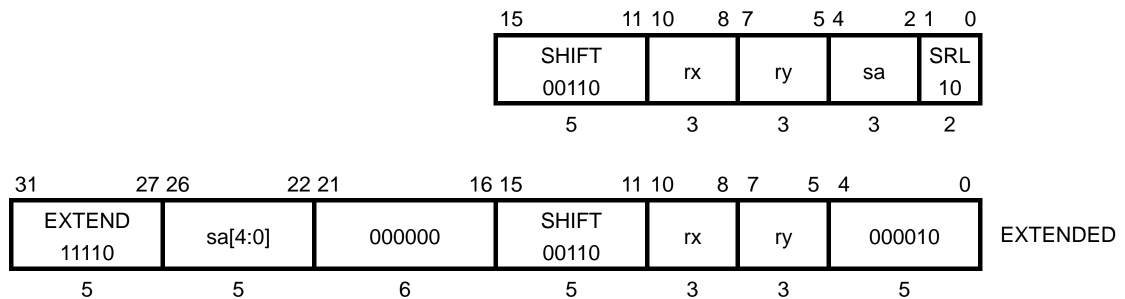
SRL *rx*, *ry*, *sa*

Shift Right Logical

Operation

$rx \Leftarrow ry >> sa$

Instruction Encoding



Description

The contents of general-purpose register *ry* is shifted right by *sa* bits. Zeros are supplied to the vacated positions on the left. The result is placed into general-purpose register *rx*. The *sa* field is only 3-bits wide. Thus the shift amount is restricted to 1 to 8. The *sa* value of 000 is defined as a shift of 8 bits.

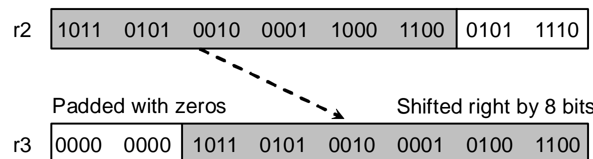
If the shift amount does not fit in the *sa* field, the instruction is EXTENDed to provide a full 5-bit field for a shift of 0 to 31.

Example

Assume that register r2 contains 0xB521_4C5E. Then, executing the instruction:

```
SRL r3,r2,8
```

places 0x00B5_214C in register r3, as shown below.



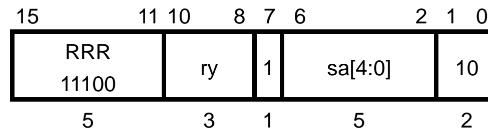
SRL *ry, sa5*

Shift Right Logical

Operation

$ry \Leftarrow ry >> sa5$

Instruction Encoding



Description

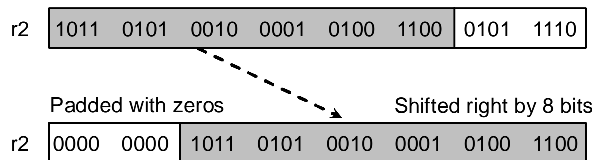
The contents of general-purpose register *ry* is shifted right by *sa* bits. Zeros are supplied to the vacated positions on the left. The result is placed back into *ry*. The *sa* field is 5-bits wide; thus the possible shift amount is 1 to 31. The *sa* value may not be 00000.

Example

Assume that register r2 contains 0xB521_4C5E. Then, executing the instruction:

```
SRL r2,8
```

places 0x00B5_214C back in register r2, as shown below.



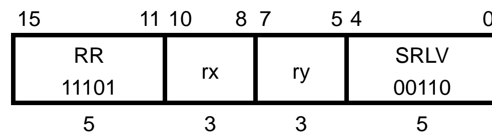
SRLV *ry*, *rx*

Shift Right Logical Variable

Operation

ry >> 5 LSBs of *rx*

Instruction Encoding



Description

The contents of general-purpose register *ry* is shifted right the number of bits specified by the five least-significant bits of general-purpose register *rx*. Zeros are supplied to the vacated positions on the left. The result is placed back into general-purpose register *ry*.

Example

None

SSUB *ry, rx, ry*

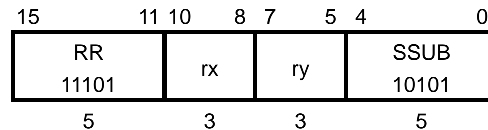
Saturated Subtract

Operation

if overflow on $rx - ry$ then $ry \Leftarrow$ 0x7FFF_FFFF ($rx \geq 0$) or 0x8000_0000 ($rx < 0$)

else $ry \Leftarrow rx - ry$

Instruction Encoding



Description

The contents of general-purpose register *ry* is subtracted from the contents of general-purpose register *rx*. On overflow, the remainder saturates to the largest representable positive number (0x7FFF_FFFF) if *rx* is zero or a positive number and to the smallest representable negative number (0x8000_0000) if *rx* is a negative number. The result is placed into *ry*. If overflow does not occur, the remainder is placed into *ry*. Both *rx* and *ry* are treated as signed integers.

An Integer Overflow exception never occurs on overflow.

Exceptions

None

SUBU *rz*, *rx*, *ry*

Subtract Unsigned

Operation

$rz \Leftarrow rx - ry$

Instruction Encoding

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

Description

The contents of general-purpose register *ry* is subtracted from the contents of general-purpose register *rx*. The remainder is placed into general-purpose register *rz*.

No Integer Overflow exception occurs under any circumstances.

Exceptions

None

SW ra, *offset* (sp)

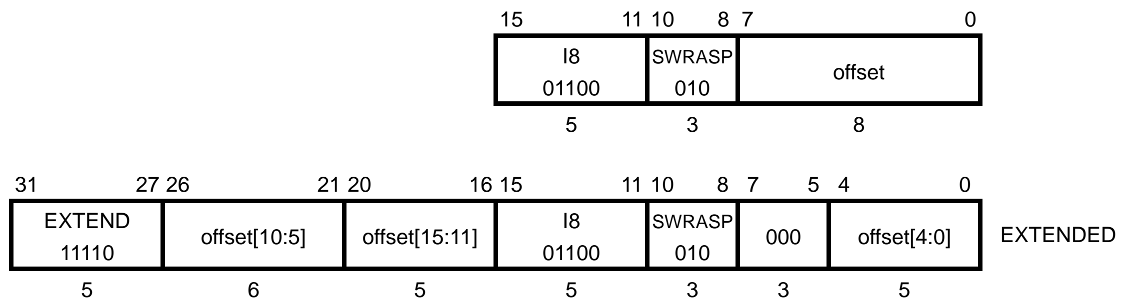
Store Word

Operation

ra = {zero-extend (*offset* || 00) + (sp)}

(EXTENDED) ra = {sign-extend (*offset*) + (sp)}

Instruction Encoding



Description

The 8-bit *immediate* offset is shifted left by two bits, zero-extended and added to the contents of stack pointer register sp (r29) to form an effective address (EA). The word in link register ra (r31) is stored at the memory location addressed by the EA.

Since the 8-bit *offset* is shifted left by two bits, the *offset* range is 0 to 1024, in increments of four. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of –32768 to +32767.

Exceptions

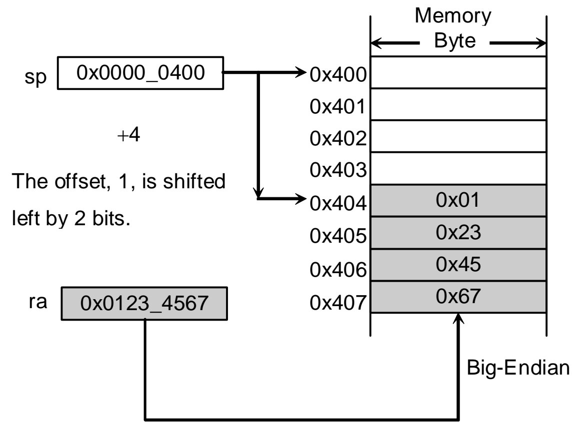
Address Error exception

Example

```
SW ra,4(sp)
```

Assume that the sp and ra registers contain 0x0000_0400 and 0x0123_4567 respectively. Since the offset value is shifted left by two bits by the processor hardware, the assembler/linker turns the specified offset (4 or binary 0100) into a code of 1 (binary 0001). Thus the instruction code for this store instruction is 0x3101.

In big-endian mode, 0x0123_4567 is stored to the memory locations at addresses 0x0404 to 0x0407.


Memory
Byte
sp
0x0000_0400
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
+4
The offset, 1, is shifted
left by 2 bits.
0x01
0x23
0x45
0x67
ra
0x0123_4567
Big-Endian

SW *ry*, *offset* (fp)

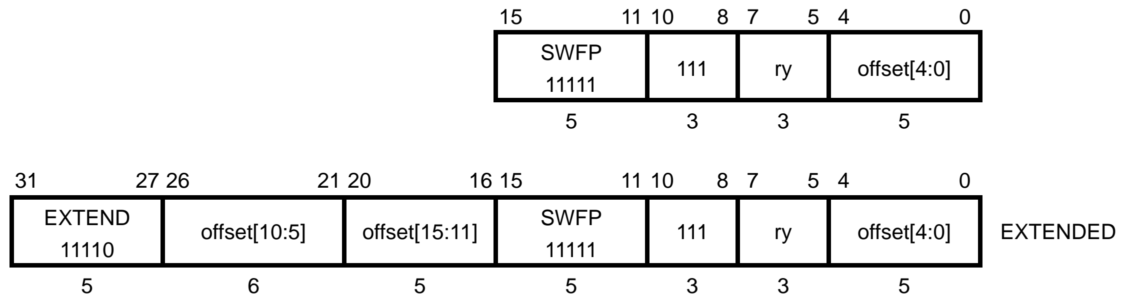
Store Word

Operation

ry = {zero-extend (*offset* || 00) + (fp)}

(EXTENDED) *ry* = {sign-extend (*offset*) + (fp)}

Instruction Encoding



Description

The 5-bit immediate *offset* is shifted left by two bits, zero-extended and added to the contents of fp register (r30) to form an effective address (EA). The word in general-purpose register *ry* is stored at the memory location addressed by the EA.

Since the 5-bit *offset* is shifted left by two bits, the offset range is 0 to 124, in increments of four. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767.

Exceptions

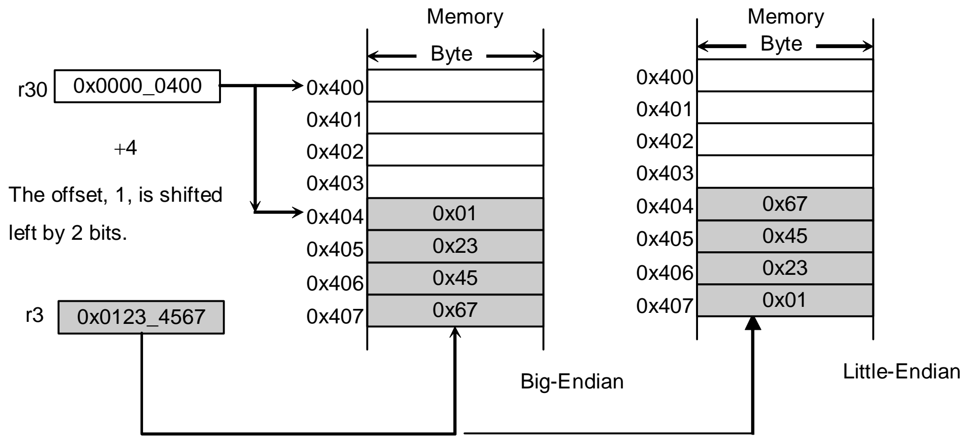
Address Error exception

Example

```
SW r3,4(fp)
```

Assume that registers fp and r3 contain 0x0000_0400 and 0x0123_4567 respectively. Since the offset value is shifted left by two bits by the processor hardware, the assembler/linker turns the specified offset (4 or binary 0100) into a code of 1 (binary 0001). Thus the instruction code for this store instruction becomes 0xFF61.

In big-endian mode, 0x123_4567 is stored to the memory locations at addresses 0x404 to 0x407. In little-endian mode, 0x6745_2301 is stored to the memory locations at addresses 0x404 to 0x407.


Memory
Byte
r30
0x0000_0400
+4
The offset, 1, is shifted
left by 2 bits.
r3
0x0123_4567
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
Big-Endian
Memory
Byte
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x67
0x45
0x23
0x01
Little-Endian

SW *rx*, *offset* (sp)

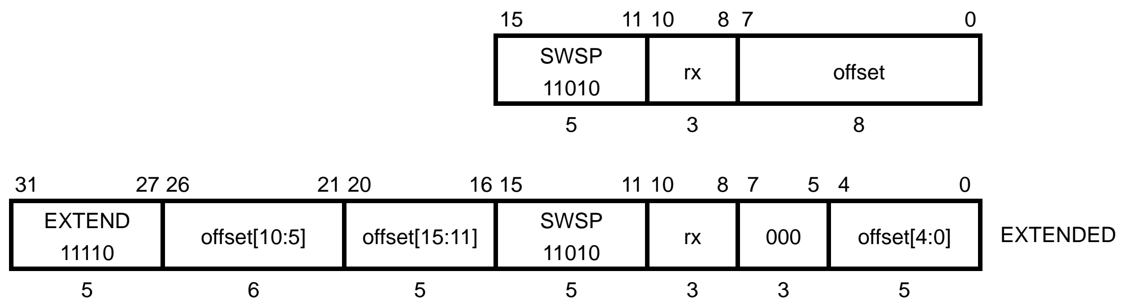
Store Word

Operation

rx = {zero-extend ($offset$ || 00) + (sp)}

(EXTENDED) rx = {sign-extend ($offset$) + (sp)}

Instruction Encoding



Description

The 8-bit *immediate* offset is shifted left by two bits, zero-extended and added to the contents of stack pointer register sp (r29) to form an effective address (EA). The word in general-purpose register *rx* is stored at the memory location addressed by the EA.

Since the 8-bit *offset* is shifted left by two bits, the offset range is 0 to 1020, in increments of four. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of –32768 to +32767.

Exceptions

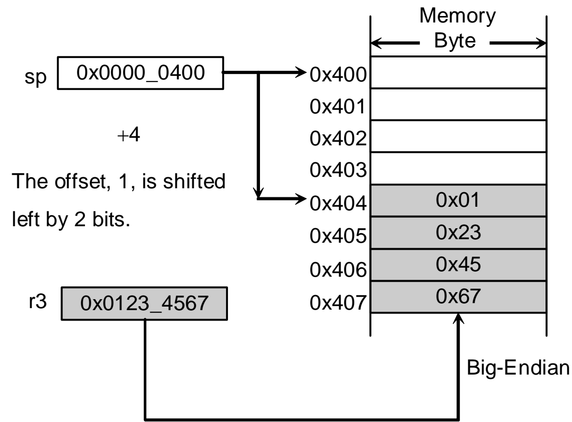
Address Error exception

Example

```
SW r3,4(sp)
```

Assume that the sp and r3 registers contain 0x0000_0400 and 0x0123_4567 respectively. Since the offset value is shifted left by two bits by the processor hardware, the assembler/linker turns the specified offset (4 or binary 0100) into a code of 1 (binary 0001). Thus the instruction code for this store instruction is 0xD301.

In big-endian mode, 0x0123_4567 is stored to the memory locations at addresses 0x404 to 0x407.


Memory
Byte
sp
0x0000_0400
+4
The offset, 1, is shifted left by 2 bits.
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
r3
0x0123_4567
Big-Endian

SW *ry*, *offset* (*base*)

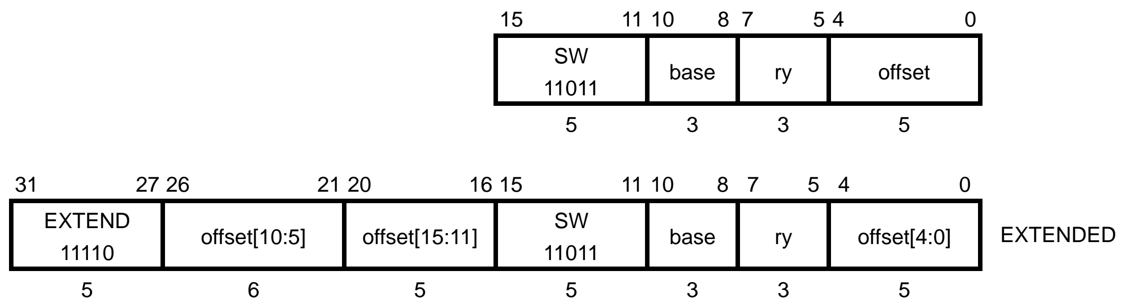
Store Word

Operation

ry = {zero-extend (*offset* || 00) + (*base*)}

(EXTENDED) *ry* = {sign-extend (*offset*) + (*base*)}

Instruction Encoding



Description

The 5-bit immediate *offset* is shifted left by two bits, zero-extended and added to the contents of general-purpose register *base* to form an effective address (EA). The word in general-purpose register *ry* is stored at the memory location addressed by the EA.

Since the 5-bit *offset* is shifted left by two bits, the offset range is 0 to 124, in increments of four. If the *offset* is outside this range, the instruction is EXTENDed to provide a 16-bit signed immediate in the range of -32768 to +32767. When EXTENDed, the *offset* operand is not shifted at all.

Exceptions

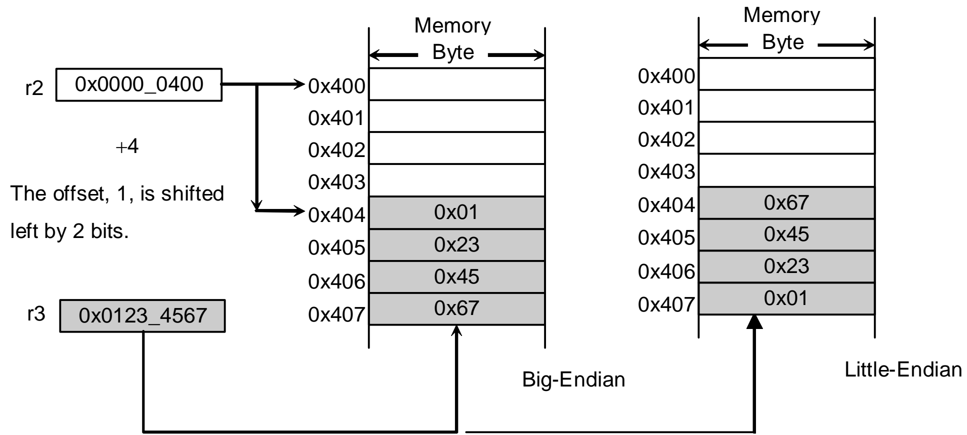
Address Error exception

Example

```
SW r3,4(r2)
```

Assume that registers r2 and r3 contain 0x0000_0400 and 0x0123_4567 respectively. Since the offset value is shifted left by two bits by the processor hardware, the assembler/linker turns the specified offset (4 or binary 0100) into a code of 1 (binary 0001). Thus the instruction code for this store instruction becomes 0xDAE1.

In big-endian mode, 0x123_4567 is stored to the memory locations at addresses 0x0404 to 0x0407. In little-endian mode, 0x6745_2301 is stored to the memory locations at addresses 0x0404 to 0x0407.


Memory
Byte
r2
0x0000_0400
+4
The offset, 1, is shifted left by 2 bits.
r3
0x0123_4567
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
Big-Endian
Memory
Byte
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x67
0x45
0x23
0x01
Little-Endian

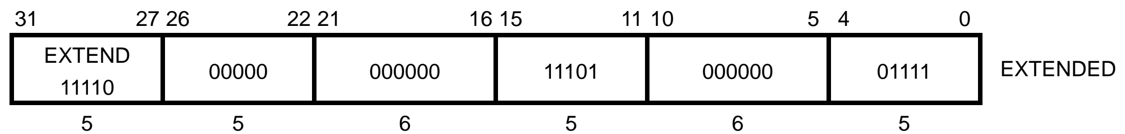
SYNC

Synchronize

Operation

メモリ同期操作

Instruction Encoding



Description

The SYNC instruction interlocks the instruction pipeline until loads and stores performed prior to the present instruction are completed before any instructions after this instruction are allowed to start. See 5.2.4, *SYNC Instruction*.

If there is no data dependency, the TX19A continues to execute subsequent instructions. This is called *non-blocking loads*. By virtue of non-blocking loads, the instruction pipeline can work on nondependent instructions.

Exceptions

None

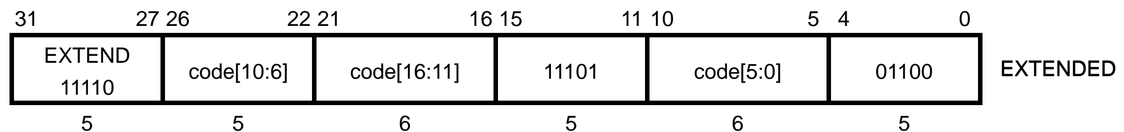
SYSCALL *code*

System Call

Operation

System call exception

Instruction Encoding



Description

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

The *code* field in a SYSCALL instruction is available for use as software parameters to pass additional information. To examine these bits, load the contents of the instruction at which the EPC register points. For details on System Call exceptions, see Section 9.1.10, *System Call Exceptions*.

Exceptions

System call exception

WAIT

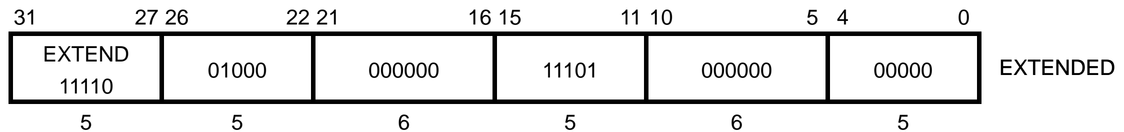
Enter Standby Mode

Operation

if Status[RP] = 1 then DOZE mode

else HALT mode

Instruction Encoding



Description

The WAIT instruction is used to freeze the instruction pipeline to reduce the processor's power consumption. If the RP bit in the Status register is set, the processor enters DOZE mode. If the RP bit is cleared, the processor enters HALT mode. See Chapter 10, *Low-Power Modes*.

The WAIT instruction must not be set in a delay slot of the branch or jump instruction.
Once the MTC0 instruction writes to the Status register, at least two instructions must be executed before the WAIT instruction. Otherwise, the operation is undefined.

Exceptions

Coprocessor Unusable exception

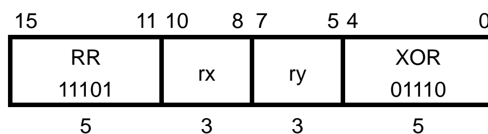
XOR *rx*, *ry*

Exclusive OR

Operation

$rx \Leftarrow rx \text{ XOR } ry$

Instruction Encoding



Description

The contents of general-purpose register *rx* is exclusive-ORed with the contents of general-purpose register *ry*. The result is placed back into general-purpose register *rx*.

Exceptions

None

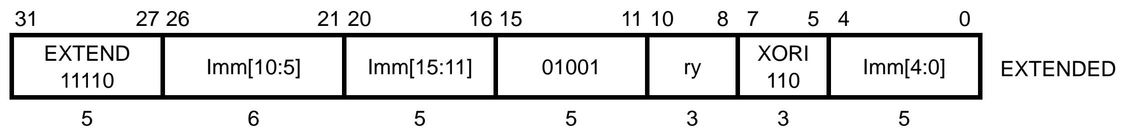
XORI *ry*, *immediate*

Exclusive OR Immediate

Operation

$ry \Leftarrow ry$ XOR $(0^{16} \parallel immediate_{15..0})$

Instruction Encoding



Description

The 16-bit *immediate* is zero-extended and exclusive-ORed with the contents of general-purpose register *ry*. The result is placed back into *ry*.

The *immediate* field is 16 bits in length. If the *immediate* size is larger than that, you need to put it in a general-purpose register and use the XOR instruction (see Section 3.3.2, *32-Bit Constants*).

Exceptions

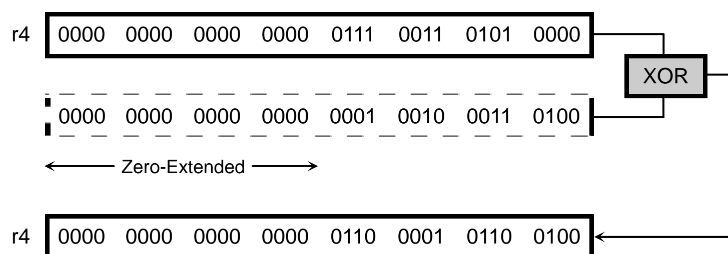
None

Example

Assume that register r4 contains 0x0000_7350. Then, executing the instruction:

```
XORI r4,0x1234
```

places 0x0000_6164 back in register r4, as shown below.



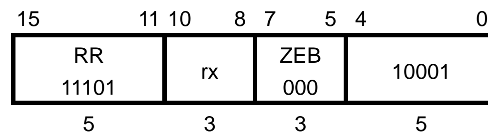
ZEB *rx*

Zero-Extend Byte

Operation

$rx \Leftarrow 0^{24} \parallel rx[7{:}0]$

Instruction Encoding



Description

The least-significant byte in general-purpose register *rx* is zero-extended. The result is placed back into *rx*.

Exceptions

None

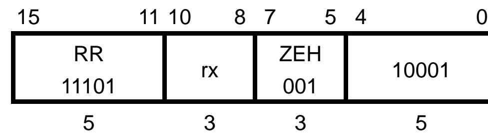
ZEH *rx*

Zero-Extend Halfword

Operation

$rx \Leftarrow 0^{16} \; || \; rx[15:0];$

Instruction Encoding



Description

The low-order halfword in general-purpose register *rx* is zero-extended. The result is placed back into *rx*.

Exceptions

None

Appendix C Programming Restrictions

In a pipelined machine like the TX19A, there are certain instructions which may disrupt the smooth operation of the pipeline due to the very pipeline structure. This appendix lists the restrictions that need to be observed in writing assembly-language programs.

C.1 32-Bit ISA Restrictions

Table C-1 Load and Store Instructions

Instructions	Restrictions
LH *rt*, *offset*(*base*) LHU *rt*, *offset*(*base*) SH *rt*, *offset*(*base*)	The target address generated by these instructions must be on a halfword boundary; i.e., it must have the least-significant bit cleared. Otherwise, an Address Error exception occurs.

LW *rt*, *offset*(*base*)
LWU *rt*, *offset*(*base*)
SW *rt*, *offset*(*base*)
	The target address generated by these instructions must be on a word boundary; i.e., it must have the two least-significant bits cleared. Otherwise, an Address Error exception occurs.

Table C-2 Jump Instructions

Instructions	Restrictions
JALR (*rd*,) *rs*	• Register *rd* may not be the same one as register *rs* because such an instruction is not restartable after the exception has been serviced. • In 32-bit ISA mode, all instructions must be word-aligned. Therefore, when jumping to a 32-bit routine, the two least-significant bits of the target register (*rs*) must be zero. Otherwise, an Address Error exception occurs when the processor fetches the instruction at the jump destination.
JR *rs*	In 32-bit ISA mode, all instructions must be word-aligned. Therefore, when jumping to a 32-bit routine, the two least-significant bits of the target register (*rs*) must be zero. Otherwise, an Address Error exception occurs when the processor fetches the instruction at the jump destination.
All jump instructions	Any jump instruction may not be in a jump or branch delay slot. The operation of the jump instruction is undefined if it is in a jump or branch delay slot.

Table C-3 Branch and Branch-Likely Instructions

Instructions	Restrictions
BGEZAL(L) *rs*, *offset* BLTZAL(L) *rs*, *offset*	Register *rs* may not be r31 because such an instruction is not restartable after the exception has been serviced.
All branch instructions	All the branch instructions may not be in a jump or branch delay slot. The operation of the branch instructions are undefined if they are in a jump or branch delay slot.

Table C-4 System Control Coprocessor (CP0) Instructions

Instructions	Restrictions
MTC0 *rt, rd* MFC0 *rt, rd* ERET WAIT	Attempts by a User-mode program to execute these instructions when the CU0 bit in the Status register is cleared causes a Coprocessor Unusable exception. Kernel mode programs can execute these instructions, regardless of the setting of the CU0 bit.
DERET	• The DERET instruction does not have a delay slot. • The operation of this instruction is undefined if the processor is not is in Debug mode (i.e., when the DM bit in the Debug register is cleared). • If you have used the MTC0 instruction to load the DEPC register with a return address, the debug exception handler must execute at least two instructions before issuing the DERET instruction.
MTC0 *rt, rd*	• Once the MTC0 instruction writes to the Status, EPC or ErrorEPC register, at least two instructions must be executed before the ERET instruction. Otherwise, contents of the register become undefined. • Once the MTC0 instruction writes to the DEPC register, at least two instructions must be executed before the DERET instruction. Otherwise, the contents of the register become undefined. • The MTC0 instruction that modifies the contents of the SSCR register must be followed by two NOPs.
WAIT ERET DERET	These instructions may not be placed in a delay slot.
WAIT	Once the MTC0 instruction writes to the Status register, at least two instructions must be executed before the WAIT instruction. Otherwise, contents of the register become undefined.

Table C-5 Special Instructions

Instructions	Restrictions
SDBBP	The SDBBP instruction may not be used within the user’s program; it is intended for use by development tools.

C.2 16-Bit ISA Restrictions

Table C-6 Load and Store Instructions

Instructions	Restrictions
LH *ry*, *offset*(*base*) LHU *ry*, *offset*(*base*) LHU *ry*, *offset*(sp) LHU *ry*, *offset*(fp) SH *ry*, *offset*(*base*) SH *ry*, *offset*(sp) SH *ry*, *offset*(fp)	The target address generated by these instructions must be on a halfword boundary; i.e., it must have the least-significant bit cleared. Otherwise, an Address Error exception occurs.

LW *ry*, *offset*(*base*)
LW *ry*, *offset*(pc)
LW *ry*, *offset*(sp)
LW *ry*, *offset*(fp)
SW *ry*, *offset*(*base*)
SW *ry*, *offset*(sp)
SW *ry*, *offset*(fp)
SW ra, *offset*(sp)
	The target address generated by these instructions must be on a word boundary; i.e., it must have the two least-significant bits cleared. Otherwise, an Address Error exception occurs.

Table C-7 Jump Instructions

Instructions	Restrictions
JALR ra, *rx* JALRC ra, *rx*	• Register *rx* may not be ra because such an instruction is not restartable after the exception has been serviced. • In 32-bit ISA mode, all instructions must be word-aligned. Therefore, when jumping to a 32-bit routine, the two least-significant bits of the target register (*rx*) must be zero. Otherwise, an Address Error exception occurs when the processor fetches the instruction at the jump destination.
JR *rx* JRC *rx*	In 32-bit ISA mode, all instructions must be word-aligned. Therefore, when jumping to a 32-bit routine, the two least-significant bits of the target register (*rx*) must be zero. Otherwise, an Address Error exception occurs when the processor fetches the instruction at the jump destination.
JR ra JRC ra	In 32-bit ISA mode, all instructions must be word-aligned. Therefore, when jumping to a 32-bit routine, the two least-significant bits of ra must be zero. Otherwise, an Address Error exception occurs when the processor fetches the instruction at the jump destination.
All jump instructions	Any jump instructions may not be in a jump delay slot.

Table C-8 Branch Instructions

Instructions	Restrictions
All branch instructions	Any branch instructions may not be in a jump delay slot.

Table C-9 Special Instructions

Instructions	Restrictions
SDBBP	The SDBBP instruction may not be used within the user's program; it is intended for use by development tools.

Table C-10 EXTENDed Instructions

Instructions	Restrictions
All EXTENDed instructions	Any EXTENDed instructions may not be in a jump delay slot.

Table C-11 System Control Coprocessor (CP0) Instructions

Instructions	Restrictions
MTC0 *rt*, *rd* MFC0 *rt*, *rd* ERET WAIT	Attempts by a User-mode program to execute these instructions when the CU0 bit in the Status register is cleared causes a Coprocessor Unusable exception. Kernel mode programs can execute these instructions, regardless of the setting of the CU0 bit.
DERET	• The DERET instruction does not have a delay slot. • The operation of this instruction is undefined if the processor is not is in Debug mode (i.e., when the DM bit in the Debug register is cleared). • If you have used the MTC0 instruction to load the DEPC register with a return address, the debug exception handler must execute at least two instructions before issuing the DERET instruction.
MTC0 *rt*, *rd*	• Once the MTC0 instruction writes to the Status, EPC or ErrorEPC register, at least two instructions must be executed before the ERET instruction. Otherwise, contents of the register become undefined. • Once the MTC0 instruction writes to the DEPC register, at least two instructions must be executed before the DERET instruction. Otherwise, contents of the register become undefined.
MFC0 *rt*, *rd* MTC0 *rt*, *rd*	The MTC0 instruction can not access the Config1, Config2, Config3, IER registers.
WAIT ERET DERET	These instructions may not be placed in a delay slot.
WAIT	Once the MTC0 instruction writes to the Status register, at least two instructions must be executed before the WAIT instruction. Otherwise, contents of the register become undefined.

Table C-12 SAVE and RESTORE Instructions

Instructions	Restrictions
All SAVE instructions All RESTORE instructions	At least one register must be specified to be saved or restored.

Appendix D Compatibility Among TX19, TX19A and TX39 Architectures

Table D-1 shows the differences between Toshiba's TX19A and TX19.

Table D-1 Comparisons Between the TX19A and the TX19

<table>
<tr><th>Feature</th><th>TX19A</th><th>TX19</th></tr>
<tr><td>Application</td><td colspan="2">Low power, high code density</td></tr>
<tr><td>Instruction Set</td><td>MIPS16e-TX</td><td>MIPS II + MIPS16 ASE</td></tr>
<tr><td>CPU Register</td><td>• 8 sets of general-purpose registers
• Same as for TX19 in all other respects</td><td>• 32 general-purpose registers
• Program counter (PC)
• Least-significant bit of the PC represents current ISA mode.
• 2 multiply-divide registers (HI/LO)</td></tr>
<tr><td>CP0 Registers</td><td colspan="2">• TX19A and TX19 assign coprocessor register numbers differently.
• TX19A complies with the MIPS32 CP0; its register numbers and bit assignments greatly differ from those of TX19.
<table>
<tr><th>Register Number</th><th>TX19A</th><th>TX19</th></tr>
<tr><td>3</td><td>—</td><td>Config</td></tr>
<tr><td>8</td><td>BadVAddr</td><td>BadVAddr</td></tr>
<tr><td>9</td><td>Count/IER</td><td>—</td></tr>
<tr><td>11</td><td>Compare</td><td>—</td></tr>
<tr><td>12</td><td>Status</td><td>Status</td></tr>
<tr><td>13</td><td>Cause</td><td>Cause</td></tr>
<tr><td>14</td><td>EPC</td><td>EPC</td></tr>
<tr><td>15</td><td>PRId</td><td>PRId</td></tr>
<tr><td>16</td><td>Config/Config 1,2,3</td><td>Debug</td></tr>
<tr><td>17</td><td>—</td><td>DEPC</td></tr>
<tr><td>22</td><td>SSCR</td><td>—</td></tr>
<tr><td>23</td><td>Debug</td><td>DESAVE</td></tr>
<tr><td>24</td><td>DEPC</td><td>—</td></tr>
<tr><td>30</td><td>ErrorEPC</td><td>—</td></tr>
<tr><td>31</td><td>DESAVE</td><td>IE</td></tr>
</table></td></tr>
<tr><td>Instruction Pipeline</td><td colspan="2">5-stage</td></tr>
<tr><td>WBU</td><td>32-bit wide, 4-deep</td><td>None</td></tr>
<tr><td>Multiply Instructions</td><td colspan="2">Latency / Execution = 2 / 1 cycles</td></tr>
<tr><td>Multiply-and-Add Instructions</td><td colspan="2">Latency / Execution = 2 / 1 cycles</td></tr>
<tr><td>Multiply-and-Subtract Instructions</td><td>Latency / Execution = 2 / 1 cycles</td><td>None</td></tr>
<tr><td>Divide Instructions</td><td colspan="2">Latency / Execution = 35 / 24 cycles
If the divide instruction is followed by an MFHI or MFLO instruction before the result is available, the pipeline stalls until the result does become available.</td></tr>
<tr><td>16-Bit ISA EXTENDed Instructions</td><td>1 cycle</td><td>2 cycles</td></tr>
<tr><td>Maskable Interrupts</td><td>• 2 software interrupts (IP[1:0])
• 3 hardware interrupts from interrupt controller (IP[4:2])
• GPR shadow sets are automatically switched, based on interrupt level.</td><td>• 4 software interrupts (Sw[3:0])
• 1 hardware interrupt from interrupt controller (7 prioritized levels)</td></tr>
</table>

<table>
<tr><th>Feature</th><th>TX19A</th><th>TX19</th></tr>
<tr><td>Exception Vector Addresses</td><td colspan="2">
<table>
<tr><th colspan="2" rowspan="2">Exception</th><th colspan="2">TX19A</th><th colspan="2">TX19</th></tr>
<tr><th>BEV = 1</th><th>BEV = 0</th><th>BEV = 1</th><th>BEV = 0</th></tr>
<tr><td colspan="2">Reset/NMI</td><td colspan="2">0xBFC0_0000</td><td colspan="2">0xBFC0_0000</td></tr>
<tr><td colspan="2" rowspan="2">Debug</td><td colspan="2">0xBFC0_0480</td><td colspan="2">0xBFC0_0200</td></tr>
<tr><td colspan="2">0xFF20_0200</td><td colspan="2">0xFF20_0200</td></tr>
<tr><td rowspan="5">Interrupts</td><td>Swi3</td><td>–</td><td>–</td><td>0xBFC0_0240</td><td>0x8000_0140</td></tr>
<tr><td>Swi2</td><td>–</td><td>–</td><td>0xBFC0_0230</td><td>0x8000_0130</td></tr>
<tr><td>Swi1</td><td rowspan="3">0xBFC0_0380
or
0xBFC0_0400</td><td rowspan="3">0x8000_0180
or
0x8000_0200</td><td>0xBFC0_0220</td><td>0x8000_0120</td></tr>
<tr><td>Swi0</td><td>0xBFC0_0210</td><td>0x8000_0110</td></tr>
<tr><td>Hardware</td><td>0xBFC0_0260</td><td>0x8000_0160</td></tr>
<tr><td colspan="2">Others</td><td>0xBFC0_0380</td><td>0x8000_0180</td><td>0xBFC0_0380</td><td>0x8000_0080</td></tr>
</table>
</td></tr>
</table>

Table D-2 gives comparisons of the 32-bit ISA of the TX19A, the TX19 and the TX39. Differences are highlighted in shaded boxes.

Table D-2 Instruction Sets of the TX19A, the TX19 and the TX39

Category	Instruction	TX19A 32-Bit ISA		TX19 32-Bit ISA		TX39	
Load/Store	Load Byte	LB	rt, offset(base)	LB	rt, offset(base)	LB	rt, offset(base)
	Load Byte Unsigned	LBU	rt, offset(base)	LBU	rt, offset(base)	LBU	rt, offset(base)
	Load Halfword	LH	rt, offset(base)	LH	rt, offset(base)	LH	rt, offset(base)
	Load Halfword Unsigned	LHU	rt, offset(base)	LHU	rt, offset(base)	LHU	rt, offset(base)
	Load Word	LW	rt, offset(base)	LW	rt, offset(base)	LW	rt, offset(base)
	Load Word Left	LWL	rt, offset(base)	LWL	rt, offset(base)	LWL	rt, offset(base)
	Load Word Right	LWR	rt, offset(base)	LWR	rt, offset(base)	LWR	rt, offset(base)
	Store Byte	SB	rt, offset(base)	SB	rt, offset(base)	SB	rt, offset(base)
	Store Halfword	SH	rt, offset(base)	SH	rt, offset(base)	SH	rt, offset(base)
	Store Word	SW	rt, offset(base)	SW	rt, offset(base)	SW	rt, offset(base)
	Store Word Left	SWL	rt, offset(base)	SWL	rt, offset(base)	SWL	rt, offset(base)
	Store Word Right	SWR	rt, offset(base)	SWR	rt, offset(base)	SWR	rt, offset(base)
	Synchronize	SYNC		SYNC		—	
ALU Immediate	Add Immediate	ADDI	rt, rs, immediate	ADDI	rt, rs, immediate	ADDI	rt, rs, immediate
	Add Immediate Unsigned	ADDIU	rt, rs, immediate	ADDIU	rt, rs, immediate	ADDIU	rt, rs, immediate
	Set On Less Than Immediate	SLTI	rt, rs, immediate	SLTI	rt, rs, immediate	SLTI	rt, rs, immediate
	Set On Less Than Immediate Unsigned	SLTIU	rt, rs, immediate	SLTIU	rt, rs, immediate	SLTIU	rt, rs, immediate
	AND Immediate	ANDI	rt, rs, immediate	ANDI	rt, rs, immediate	ANDI	rt, rs, immediate
	OR Immediate	ORI	rt, rs, immediate	ORI	rt, rs, immediate	ORI	rt, rs, immediate
	Exclusive-OR Immediate	XORI	rt, rs, immediate	XORI	rt, rs, immediate	XORI	rt, rs, immediate
	Load Upper Immediate	LUI	rt, rs, immediate	LUI	rt, rs, immediate	LUI	rt, rs, immediate
2/3-Operand Register Type	Add	ADD	rd, rs, rt	ADD	rd, rs, rt	ADD	rd, rs, rt
	Add Unsigned	ADDU	rd, rs, rt	ADDU	rd, rs, rt	ADDU	rd, rs, rt
	Subtract	SUB	rd, rs, rt	SUB	rd, rs, rt	SUB	rd, rs, rt
	Subtract Unsigned	SUBU	rd, rs, rt	SUBU	rd, rs, rt	SUBU	rd, rs, rt
	Set On Less Than	SLT	rd, rs, rt	SLT	rd, rs, rt	SLT	rd, rs, rt
	Set On Less Than Unsigned	SLTU	rd, rs, rt	SLTU	rd, rs, rt	SLTU	rd, rs, rt
	AND	AND	rd, rs, rt	AND	rd, rs, rt	AND	rd, rs, rt
	OR	OR	rd, rs, rt	OR	rd, rs, rt	OR	rd, rs, rt
	Exclusive-OR	XOR	rd, rs, rt	XOR	rd, rs, rt	XOR	rd, rs, rt
	NOR	NOR	rd, rs, rt	NOR	rd, rs, rt	NOR	rd, rs, rt
	Count Leading Ones in Word	CLO	rd, rs	—		—	
	Count Leading Zeros in Word	CLZ	rd, rs	—		—	
	Move Conditional on Not Zero	MOVN	rd, rs, rt	—		—	
	Move Conditional on Zero	MOVZ	rd, rs, rt	—		—	

Category	Instruction	TX19A 32-Bit ISA		TX19 32-Bit ISA		TX39	
Shift	Shift Left Logical	SLL	rd, rs, ra	SLL	rd, rs, ra	SLL	rd, rs, ra
	Shift Left Logical Variable	SLLV	rd, rs, rt	SLLV	rd, rs, rt	SLLV	rd, rs, rt
	Shift Right Logical	SRL	rd, rs, sa	SRL	rd, rs, sa	SRL	rd, rs, sa
	Shift Right Logical Variable	SRLV	rd, rs, rt	SRLV	rd, rs, rt	SRLV	rd, rs, rt
	Shift Right Arithmetic	SRA	rd, rs, sa	SRA	rd, rs, sa	SRA	rd, rs, sa
	Shift Right Arithmetic Variable	SRAV	rd, rs, rt	SRAV	rd, rs, rt	SRAV	rd, rs, rt
Multiply, Divide, Multiply-and-Add and Multiply-and-Subtract	Multiply	MUL	rd, rs, rt	—		—	
		MULT	rs, rt	MULT	rs, rt	MULT	rs, rt
		MULT	rd, rs, rt	MULT	rd, rs, rt	MULT	rd, rs, rt
	Multiply Unsigned	MULTU	rs, rt	MULTU	rs, rt	MULTU	rs, rt
		MULTU	rd, rs, rt	MULTU	rd, rs, rt	MULTU	rd, rs, rt
	Divide	DIV	rs, rt	DIV	rs, rt	DIV	rs, rt
	Divide Unsigned	DIVU	rs, rt	DIVU	rs, rt	DIVU	rs, rt
	Move From HI	MFHI	rd	MFHI	rd	MFHI	rd
	Move From LO	MFLO	rd	MFLO	rd	MFLO	rd
	Move To HI	MTHI	rd	MTHI	rd	MTHI	rd
	Move To LO	MTLO	rd	MTLO	rd	MTLO	rd
	Multiply-and-Add	MADD	rs, rt	MADD	rs, rt	MADD	rs, rt
		MADD	rd, rs, rt	MADD	rd, rs, rt	MADD	rd, rs, rt
	Multiply-and-Add Unsigned	MADDU	rs, rt	MADDU	rs, rt	MADDU	rs, rt
		MADDU	rd, rs, rt	MADDU	rd, rs, rt	MADDU	rd, rs, rt
	Multiply and Subtract	MSUB	rs, rt	—		—	
		MSUB	rd, rs, rt	—		—	
	Multiply and Subtract Unsigned	MSUBU	rs, rt	—		—	
		MSUBU	rd, rs, rt	—		—	
Jump	Jump	J	target	J	target	J	target
	Jump And Link	JAL	target	JAL	target	JAL	target
	Jump and Link exchange	JALX	target	JALX	target	—	
	Jump Register	JR	rs	JR	rs	JR	rs
	Jump And Link Register	JALR	(rd), rs	JALR	(rd), rs	JALR	(rd), rs
Branch	Unconditional Branch	B	offset	—		—	
	Branch On Equal	BEQ	rs, rt, offset	BEQ	rs, rt, offset	BEQ	rs, rt, offset
	Branch On Not Equal	BNE	rs, rt, offset	BNE	rs, rt, offset	BNE	rs, rt, offset
	Branch On Greater Than Zero	BGTZ	rs, offset	BGTZ	rs, offset	BGTZ	rs, offset
	Branch On Greater Than Zero or Equal to Zero	BGEZ	rs, offset	BGEZ	rs, offset	BGEZ	rs, offset
	Branch On Less Than Zero	BLTZ	rs, offset	BLTZ	rs, offset	BLTZ	rs, offset
	Branch On Less Than Zero or Equal to Zero	BLEZ	rs, offset	BLEZ	rs, offset	BLEZ	rs, offset
	Branch On Less Than Zero And Link	BLTZAL	rs, offset	BLTZAL	rs, offset	BLTZAL	rs, offset
	Branch On Greater Than Zero And Link	BGEZAL	rs, offset	BGEZAL	rs, offset	BGEZAL	rs, offset

Category	Instruction	TX19A 32-Bit ISA	TX19 32-Bit ISA	TX39
Branch-Likely	Branch And Link	BAL offset	—	—
	Branch On Equal Likely	BEQL rs, rt, offset	BEQL rs, rt, offset	BEQL rs, rt, offset
	Branch On Not Equal Likely	BNEL rs, rt, offset	BNEL rs, rt, offset	BNEL rs, rt, offset
	Branch On Greater Than Zero Likely	BGTZL rs, offset	BGTZL rs, offset	BGTZL rs, offset
	Branch On Greater Than Zero or Equal to Zero Likely	BGEZL rs, offset	BGEZL rs, offset	BGEZL rs, offset
	Branch On Less Than Zero Likely	BLTZL rs, offset	BLTZL rs, offset	BLTZL rs, offset
	Branch On Less Than Zero or Equal to Zero Likely	BLEZL rs, offset	BLEZL rs, offset	BLEZL rs, offset
	Branch On Less Than Zero And Link Likely	BLTZALL rs, offset	BLTZALL rs, offset	BLTZALL rs, offset
	Branch On Greater Than Zero And Link Likely	BGEZALL rs, offset	BGEZALL rs, offset	BGEZALL rs, offset
Trap	Trap If Equal	TEQ rs, rt	—	—
	Trap If Equal Immediate	TEQI rs, Immediate	—	—
	Trap If Greater Than or Equal	TGE rs, rt	—	—
	Trap If Greater Than or Equal Immediate	TGEI rs, Immediate	—	—
	Trap If Greater Than or Equal Unsigned	TGEU rs, rt	—	—
	Trap If Greater Than or Equal Immediate Unsigned	TGEIU rs, Immediate	—	—
	Trap If Less Than	TLT rs, rt	—	—
	Trap If Less Than Immediate	TLTI rs, Immediate	—	—
	Trap If Less Than Unsigned	TLTU rs, rt	—	—
	Trap If Less Than Immediate Unsigned	TLTIU rs, Immediate	—	—
	Trap If Not Equal	TNE rs, rt	—	—
	Trap If Not Equal Immediate	TNEI rs, Immediate	—	—
Coprocessor	Move To Coprocessor	—	MTCz rt, rd	MTCz rt, rd
	Move From Coprocessor	—	MFCz rt, rd	MFCz rt, rd
	Move Control To Coprocessor	—	CTCz rt, rd	CTCz rt, rd
	Move Control From Coprocessor	—	CFCz rt, rd	CFCz rt, rd
	Coprocessor Operation	—	COPz cofun	COPz cofun
	Branch On Coprocessor z True	—	BCzT offset	BCzT offset
	Branch On Coprocessor z True Likely	—	BCzTL offset	BCzTL offset
	Branch On Coprocessor z False	—	BCzF offset	BCzF offset
	Branch On Coprocessor z False Likely	—	BCzFL offset	BCzFL offset

Category	Instruction	TX19A 32-Bit ISA	TX19 32-Bit ISA	TX39
System Control Coprocessor	Move To CP0	MTC0 rt, rd	MTC0 rt, rd	MTC0 rt, rd
	Move From CP0	MFC0 rt, rd	MFC0 rt, rd	MFC0 rt, rd
	Restore From Exception	—	RFE	RFE
	Exception Return	ERET	—	—
	Debug Exception Return	DERET	DERET	DERET
	Cache	—	CACHE op, offset(base)	CACHE op, offset(base)
	Read Indexed TLB Entry (*1)	—	(TLBR)	(TLBR)
	Write Indexed TLB Entry (*1)	—	(TLBWI)	(TLBWI)
	Write Random TLB Entry (*1)	—	(TLBWR)	(TLBWR)
	Probe TLB for Matching Entry (*1)	—	(TLBP)	(TLBP)
Special	System Call	SYSCALL code	SYSCALL code	SYSCALL code
	Breakpoint	BREAK code	BREAK code	BREAK code
	Software Debug Breakpoint Exception	SDBBP code	SDBBP code	SDBBP code
	Enter Standby Mode	WAIT	—	—

(*1) Treated as a NOP.

Table D-3 gives comparisons of the 16-bit ISA of the TX19A and the TX19, and the MIPS16 ASE.

Table D-3 Instruction Sets of the TX19A 16-bit ISA, the TX19 16-bit ISA and the MIPS16 ASE

Category	Instruction	TX19A16-Bit ISA	TX19 16-Bit ISA	MIPS16 ASE
Load/Store	Load Byte	LB ry, offset(base)	LB ry, offset(base)	LB ry, offset(base)
	Load Byte Unsigned	LBU ry, offset(base)	LBU ry, offset(base)	LBU ry, offset(base)
		LBU ry, offset(sp)	—	—
		LBU ry, offset(fp)	—	—
	Load Halfword	LH ry, offset(base)	LH ry, offset(base)	LH ry, offset(base)
	Load Halfword Unsigned	LHU ry, offset(base)	LHU ry, offset(base)	LHU ry, offset(base)
		LHU ry, offset(sp)	—	—
		LHU ry, offset(fp)	—	—
	Load Word	LW ry, offset(base)	LW ry, offset(base)	LW ry, offset(base)
		LW ry, offset(pc)	LW ry, offset(pc)	LW ry, offset(pc)
		LW ry, offset(sp)	LW ry, offset(sp)	LW ry, offset(sp)
		LW ry, offset(fp)	—	—
	Load Word Unsigned	—	—	LWU ry, offset(sp)
	Load Doubleword	—	—	LD ry, offset(base)
		—	—	LD ry, offset(pc)
		—	—	LD ry, offset(sp)
	Store Byte	SB ry, offset(base)	SB ry, offset(base)	SB ry, offset(base)
		SB ry, offset(sp)	—	—
		SB ry, offset(fp)	—	—
	Store Halfword	SH ry, offset(base)	SH ry, offset(base)	SH ry, offset(base)
		SH ry, offset(sp)	—	—
		SH ry, offset(fp)	—	—
	Store Word	SW ry, offset(base)	SW ry, offset(base)	SW ry, offset(base)
		SW ry, offset(sp)	SW ry, offset(sp)	SW ry, offset(sp)
		SW ra, offset(sp)	SW ra, offset(sp)	SW ra, offset(sp)
		SW ry, offset(fp)	—	—
	Store Doubleword	—	—	SD ry, offset(base)
		—	—	SD ry, offset(pc)
		—	—	SD ry, offset(sp)
	Synchronize	SYNC	—	—

Category	Instruction	TX19A 16-Bit ISA	TX19 16-Bit ISA	MIPS16 ASE
ALU Immediate	Add Immediate	ADDIU ry, rx, immediate	ADDIU ry, rx, immediate	ADDIU ry, rx, immediate
		ADDIU rx, immediate	ADDIU rx, immediate	ADDIU rx, immediate
		ADDIU sp, immediate	ADDIU sp, immediate	ADDIU sp, immediate
		ADDIU rx, pc, immediate	ADDIU rx, pc, immediate	ADDIU rx, pc, immediate
		ADDIU rx, sp, immediate	ADDIU rx, sp, immediate	ADDIU rx, sp, immediate
		ADDIU fp, immediate	—	—
	Doubleword Add Immediate	—	—	DADDIU ry, rx, immediate
		—	—	DADDIU ry, immediate
		—	—	DADDIU sp, immediate
		—	—	DADDIU ry, pc, immediate
		—	—	DADDIU ry, sp, immediate
	Set On Less Than Immediate	SLTI rx, immediate	SLTI rx, immediate	SLTI rx, immediate
	Set On Less Than Immediate Unsigned	SLTIU rx, immediate	SLTIU rx, immediate	SLTIU rx, immediate
	Compare Immediate	CMPI rx, immediate	CMPI rx, immediate	CMPI rx, immediate
	Load Immediate	LI rx, immediate	LI rx, immediate	LI rx, immediate
	Logical AND Immediate	ANDI rx, immediate	—	—
	Logical OR Immediate	ORI rx, immediate	—	—
	Logical Exclusive-OR Immediate	XORI rx, immediate	—	—
	Load Upper Immediate	LUI rx, immediate	—	—
2/3-Operand Register Type	Add Unsigned	ADDU rz, rx, ry	ADDU rz, rx, ry	ADDU rz, rx, ry
	Doubleword Add Unsigned	—	—	DADDU rz, rx, ry
	Subtract Unsigned	SUBU rz, rx, ry	SUBU rz, rx, ry	SUBU rz, rx, ry
	Doubleword Subtract Unsigned	—	—	DSUBU rz, rx, ry
	Set On Less Than	SLT rx, ry	SLT rx, ry	SLT rx, ry
	Set On Less Than Unsigned	SLTU rx, ry	SLTU rx, ry	SLTU rx, ry
	Compare	CMP rx, ry	CMP rx, ry	CMP rx, ry
	Negate	NEG rx, ry	NEG rx, ry	NEG rx, ry
	AND	AND rx, ry	AND rx, ry	AND rx, ry
	OR	OR rx, ry	OR rx, ry	OR rx, ry
	Exclusive-OR	XOR rx, ry	XOR rx, ry	XOR rx, ry
	Not	NOT rx, ry	NOT rx, ry	NOT rx, ry
	Move	MOVE ry, r32	MOVE ry, r32	MOVE ry, r32
		MOVE r32, rz	MOVE r32, rz	MOVE r32, rz
		MOVE fp, r32	—	—
	Bit Search One Forward	BS1F ry, rx	—	—
	Bit Field Insert	BFINS ry, rx, bit2, bit1	—	—
	Maximum Signed	MAX rz, rx, ry	—	—
	Minimum Signed	MIN rz, rx, ry	—	—
	Sign-Extend Byte	SEB rx	—	—
	Sign-Extend Halfword	SEH rx	—	—
	Zero-Extend Byte	ZEB rx	—	—
	Zero-Extend Halfword	ZEH rx	—	—

Category	Instruction	TX19A 16-Bit ISA	TX19 16-Bit ISA	MIPS16 ASE
Shift	Shift Left Logical	SLL rx, ry, sa	SLL rx, ry, sa	SLL rx, ry, sa
		SLL ry, sa5	—	—
	Shift Left Logical Variable	SLLV ry, rx	SLLV ry, rx	SLLV ry, rx
	Shift Right Logical	SRL rx, ry, sa	SRL rx, ry, sa	SRL rx, ry, sa
		SRL ry, sa5	—	—
	Shift Right Logical Variable	SRLV ry, rx	SRLV ry, rx	SRLV ry, rx
	Shift Right Arithmetic	SRA rx, ry, sa	SRA rx, ry, sa	SRA rx, ry, sa
		SRA ry, sa5	—	—
	Shift Right Arithmetic Variable	SRAV ry, rx	SRAV ry, rx	SRAV ry, rx
	Doubleword Shift Left Logical	—	—	DSLL rx, ry, sa
	Doubleword Shift Left Logical Variable	—	—	DSLLV ry, rx
	Doubleword Shift Right Logical	—	—	DSRL rx, ry, sa
	Doubleword Shift Right Logical Variable	—	—	DSRLV ry, rx
	Doubleword Shift Right Arithmetic	—	—	DSRA rx, ry, sa
	Doubleword Shift Right Arithmetic Variable	—	—	DSRAV ry, rx
SAVE/ RESTORE	SAVE	SAVE reg_list3, framesize4	—	—
		SAVE reg_list3, xsregs, aregs, framesize8	—	—
	RESTORE	RESTORE reg_list3, framesize4	—	—
		RESTORE reg_list3, xsregs, aregs, framesize8	—	—
Multiply, Divide and Multiply-and-Add	Mutiply	MULT rx, ry	MULT rx, ry	MULT rx, ry
		MULT ry, rx, ry	—	—
	Multiply Unsigned	MULTU rx, ry	MULTU rx, ry	MULTU rx, ry
		MULTU ry, rx, ry	—	—
	Doubleword Mutiply	—	—	DMULT rx, ry
	Doubleword Multiply Unsigned	—	—	DMULTU rx, ry
	Mutiply And Add	MADD rx, ry	—	—
	Mutiply And Add Unsigned	MADDU rx, ry	—	—
	Saturated Add	SADD ry, rx, ry	—	—
	Saturated Subtract	SSUB ry, rx, ry	—	—
	Divide	DIV rx, ry	DIV rx, ry	DIV rx, ry
	Divide Unsigned	DIVU rx, ry	DIVU rx, ry	DIVU rx, ry
	Doubleword Divide	—	—	DDIV rx, ry
	Doubleword Divide Unsigned	—	—	DDIVU rx, ry
	Divide Exception	DIVE rx, ry	—	—
	Divide Exception Unsigned	DIVEU rx, ry	—	—
	Move From HI	MFHI rx	MFHI rx	MFHI rx
	Move From LO	MFLO rx	MFLO rx	MFLO rx
	Move To HI	MTHI rx	—	—
	Move To LO	MTLO rx	—	—

Category	Instruction	TX19A 16-Bit ISA	TX19 16-Bit ISA	MIPS16 ASE
Bit Manipulation	Bit Test	BTST offset(base3), pos3	—	—
		BTST offset(r0), pos3	—	—
		BTST offset(fp), pos3	—	—
	Bit Extract	BEXT offset(base3), pos3	—	—
		BEXT offset(r0), pos3	—	—
		BEXT offset(fp), pos3	—	—
	Bit Clear	BCLR offset(base3), pos3	—	—
		BCLR offset(r0), pos3	—	—
		BCLR offset(fp), pos3	—	—
	Bit Set	BSET offset(base3), pos3	—	—
		BSET offset(r0), pos3	—	—
		BSET offset(fp), pos3	—	—
	Bit Insert	BINS offset(base3), pos3	—	—
		BINS offset(r0), pos3	—	—
		BINS offset(fp), pos3	—	—
	Add Immediate to Memory Word	ADDMIU offset(base3), imm3	—	—
		ADDMIU offset(r0), imm3	—	—
Coprocessor	Move To Coprocessor 0	MTC0 rx, cp0rd32	—	—
	Move From Coprocessor 0	MFC0 ry, cp0rs32	—	—
	Add Coprocessor 0 Immediate Unsigned	AC0IU cp0rt32, imm3	—	—
Jump	Jump And Link	JAL target	JAL target	JAL target
	Jump And Link exchange	JALX target	JALX target	JALX target
	Jump Register	JR rx	JR rx	JR rx
		JR ra	JR ra	JR ra
	Jump Register Compact	JRC rx	—	—
		JRC ra	—	—
	Jump And Link Register	JALR ra, rx	JALR ra, rx	JALR ra, rx
	Jump And Link Register Compact	JALRC ra, rx	—	—
Branch	Branch On Equal To Zero	BEQZ rx, offset	BEQZ rx, offset	BEQZ rx, offset
	Branch On Not Equal To Zero	BNEZ rx, offset	BNEZ rx, offset	BNEZ rx, offset
	Branch On T8 Equal To Zero	BTEQZ offset	BTEQZ offset	BTEQZ offset
	Branch On T8 Not Equal To Zero	BTNEZ offset	BTNEZ offset	BTNEZ offset
	Branch Unconditional	B offset	B offset	B offset
	Branch And Link	BAL offset	—	—
Special	Breakpoint	BREAK code	BREAK code	BREAK code
	Software Debug Breakpoint Exception	SDBBP code	SDBBP code	SDBBP code
	Extend	EXTEND immediate	EXTEND immediate	EXTEND immediate
	Disable Interrupt	DI	—	—
	Enable Interrupt	EI	—	—
	System Call	SYSCALL code	—	—
	Exception Return	ERET	—	—
	Debug Exception Return	DERET	—	—
	Enter Standby Mode	WAIT	—	—

Appendix E 32-Bit ISA Instruction Bit Encodings

This appendix shows the encoding used for the 32-bit ISA.

Table E-1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Opcodes marked with this symbol are reserved for future use. Executing such an instruction causes a Reserved Instruction Exception.
β	Executing instructions marked with this symbol causes a Reserved Instruction Exception.
θ	Executing instructions with an opcode marked with this symbol causes a Coprocessor Unusable Exception or a Reserved Instruction Exception.
σ	Opcodes marked with this symbol represent an EJTAG support instruction. If EJTAG is not implemented, executing such an instruction causes a Reserved Instruction Exception.

Table 0-1 MIPS32 Encoding of the Opcode Field

opcode		bits 28..26							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	*SPECIAL*	*REGIMM*	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0*	(*COP1*) θ	(*COP2*) θ	(*COP3*) θ	BEQL	BNEL	BLEZL	BGTZL
3	011	β	β	β	β	*SPECIAL2*	JALX	β	*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	β
5	101	SB	SH	SWL	SW	β	β	SWR	(CACHE)
6	110	(LL)	(LWC1) β	(LWC2) β	(PREF)	β	(LDC1) β	(LDC2) β	β
7	111	(SC)	(SWC1) β	(SWC2) β	*	β	(SDC1) β	(SDC2) β	β

Table 0-2 MIPS32 *SPECIAL* Opcode Encoding of Function Field

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL	β	SRL	SRA	SLLV	*	SRLV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	β	*	β	β
3	011	MULT	MULTU	DIV	DIVU	β	β	β	β
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	β	β	β	β
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	β	*	β	β	β	*	β	β

Table 0-3 MIPS32 *REGIMM* Encoding of rt Field

rt		bits 18..16							
		0	1	2	3	4	5	6	7
bits 20..19		000	001	010	011	100	101	110	111
0	00	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	01	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	10	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	11	*	*	*	*	*	*	*	*

Table 0-4 MIPS32 *SPECIAL2* Encoding of Function Field

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	θ	MSUB	MSUBU	θ	θ
1	001	θ	θ	θ	θ	θ	θ	θ	θ
2	010	θ	θ	θ	θ	θ	θ	θ	θ
3	011	θ	θ	θ	θ	θ	θ	θ	θ
4	100	CLZ	CLO	θ	θ	β	β	θ	θ
5	101	θ	θ	θ	θ	θ	θ	θ	θ
6	110	θ	θ	θ	θ	θ	θ	θ	θ
7	111	θ	θ	θ	θ	θ	θ	θ	SDBBPσ

Table 0-5 MIPS32 *COP0* Encoding of rs Field

rs		bits 23..21							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC0	β	*	*	MTC0	β	*	*
1	01	*	*	*	*	*	*	*	*
2	10	*CO*							
3	11								

Table 0-6 MIPS32 *COP0* Encoding of Function Field When rs = *CO*

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	(TLBR)	(TLBWI)	*	*	*	(TLBWR)	*
1	001	(TLBP)	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	ERET	*	*	*	*	*	*	DERETσ
4	100	WAIT	*	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Appendix F 16-Bit ISA Instruction Bit Encodings

This appendix shows the encoding used for the 16-bit ISA. Opcodes marked with * cause an Reserved Instruction exception.

Table F-1 Major Opcode Map

	Instruction Bits [13:11]							
[15:14]	000	001	010	011	100	101	110	111
00	addiusp	addiupc	b	***JAL(X)***	beqz	bnez	***SHIFT***	***FP-B***
01	***RRI-A***	***addiu8***	slti	sltiu	***I8***	li	cmpi	***SP-B***
10	lb	lh	lwsp	lw	lbu	lhu	lwpc	***FP-SP-H***
11	sb	sh	swsp	sw	***RRR***	***RR***	**extend**	***SPECIAL***

Table F-2 *JAL(X)* Minor Opcode Map

Instruction Bit [26]	
0	1
jal	jalx

Table F-3 *FP-B* and extend + *FP-B* Minor Opcode Map

Instruction Bit [7]	
0	1
lbfp	sbfp

Table F-4 *SP-B* and extend + *SP-B* Minor Opcode Map

Instruction Bit [7]	
0	1
lbsp	sbsp

Table F-5 *FP-SP-H* and extend + *FP-SP-H* Minor Opcode Map

	Instruction Bit [0]	
[7]	0	1
0	lhsp	lhfp
1	shsp	shfp

Table F-6 *SPECIAL* and extend + *SPECIAL* Minor Opcode Map

Instruction Bits [10:8]							
000	001	010	011	100	101	110	111
btst	bclr	bset	bins	bal	bext	lwfp	swfp

Table F-7 extend + *addiu8* Minor Opcode Map

Instruction Bits [7:5]							
000	001	010	011	100	101	110	111
addiu8	*	addmiu	*	andi	ori	xori	lui

Table F-8 *RRI-A* Minor Opcode Map

Instruction Bit [4]	
0	1
addiu	*

Table F-9 *I8* and extend + *I8* Minor Opcode Map

Instruction Bits [10:8]							
000	001	010	011	100	101	110	111
bteqz	btnez	swrasp	adjsp	***SVRS***	mov32r	adjfp	movr32

Table F-10 *I8* + *SVRS* and extend + *I8* + *SVRS* Minor Opcode Map

Instruction Bits [7]	
0	1
restore	save

Table F-11 *RRR* Minor Opcode Map

	Instruction Bits [1:0]			
[7]	00	01	10	11
0	ac0iu	addu	sra/mthi	subu
1	sll/***INT***		srl/mtlo	

Note: mthi, mtlo and ***INT*** must have instruction bits [6:2] cleared.

Table F-12 *RRR* + *INT* Minor Opcode Map

Instruction Bits [10:8]							
000	001	010	011	100	101	110	111
di	ei	*	*	*	*	*	*

Table F-13 *SHIFT* Minor Opcode Map

	Instruction Bits [1:0]			
[2]	00	01	10	11
0	sll	mfc0	srl	sra
1	sll	mtc0	srl	sra

Table F-14 *RR* Minor Opcode Map

	Instruction Bits [2:0]							
[4:3]	000	001	010	011	100	101	110	111
00	***J*(*AL*)*R*(*C*)**	sdbbp	slt	sltu	sllv	break	srlv	srav
01	movfp	*	cmp	neg	and	or	xor	not
10	mfhi	***CNVT***	mflo	*	sadd	ssub	madd	maddu
11	mult	multu	div	divu	mult	multu	dive	diveu

Table F-15 *RR* + *J*(*AL*)*R*(*C*) Minor Opcode Map

Instruction Bits [7:5]							
000	001	010	011	100	101	110	111
jr rs	jr ra	jalr ra,rs	*	jrc rs	jrc ra	jalrc ra,rs	*

Table F-16 *RR* + *CNVT* Minor Opcode Map

Instruction Bits [7:5]							
000	001	010	011	100	101	110	111
zeb	zeh	*	*	seb	seh	*	*

Table F-17 extend + *RR* Minor Opcode Map

	Instruction Bits [2:0]							
[4:3]	000	001	010	011	100	101	110	111
00	wait	(tlbr)	(tlbwi)	*	*	***MAX/MIN***	(tlbwr)	***BS1F/BFINS***
01	(tlbp)	*	*	*	syscall	*	*	sync
10	*	*	*	*	*	*	*	(cache)
11	eret	*	*	*	*	*	*	deret

Table F-18 extend + *RR* + *MAX/MIX* Minor Opcode Map

Instruction Bit [26]	
0	1
max	min

Table F-19 extend + *RR* + *BS1F/BFINS* Minor Opcode Map

Instruction Bit [26]	
0	1
bfins	bs1f