Chapter 1 Introduction

This chapter is useful for readers who want a general understanding of the features of the TX19. This chapter also provides a general description of how the TX19 RISC design differs from such CISC processors as the 900/L1 from Toshiba.

1.1 Processor General Features

The TX19 is a family of high-performance, compact core microprocessors that offer the speed of a 32-bit RISC solution with the added advantage of a significantly reduced code size and low-power performance of a 16-bit architecture. The instruction set of the TX19 includes as a subset the 32-bit instructions of the TX39, which is based on MIPS Technologies, Inc.'s R3000A architecture. Thus the TX19 preserves software compatibility forward from the TX39. Additionally, the TX19 supports the MIPS16 Application-Specific Extensions (ASE) for improved code density.

The TX19 family of integrated processors and controllers is built on an TX19 core processor, an on-chip bus and a selection of intelligent peripherals appropriate for specific applications. The TX19 is available both as an ASIC-ready core and in a family of standard ASSP products.

16-Bit and 32-Bit ISA Modes

- The 16-bit instructions are object-code compatible with the MIPS16 ASE.
  Note: The TX19 does not provide support for MIPS16 instructions for 64-bit operations.
- The 32-bit instructions are object-code compatible with the high-performance TX39 family.
  - Efficient run-time switching between 16-bit and 32-bit ISA modes through an instruction
  - Upward compatible with the MIPS R3000A except for some of the coprocessor and TLB instructions
  - Hardware interlocks: The instruction immediately following a load can use the contents of the loaded register, eliminating the need to insert a NOP (No Operation) instruction.
  - Branch-likely instructions allow the processor to execute the instruction immediately following the branch while the target instruction is being fetched. 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.
- 5-stage pipeline
- Provisions for independent on-chip instruction and data memory with an access time of one clock cycle
- Provisions for independent on-chip instruction and data caches
- Provisions for an on-chip write buffer
- Harvard architecture
  The TX19 uses separate buses for code and data operands. In the TX19, 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: Executes the next useful instruction in a load delay slot in the event a load from external memory causes a large latency.
- On-chip multiplier/accumulator (MAC): Executes 32-bit x 32-bit multiplier operations with a 64-bit accumulation in a single clock cycle.
- 4-Gbyte virtual address space
- Provides support for 4 coprocessors: The TX19 contains the system control coprocessor (CP0) for system configuration, exception handling and memory management.

Low Power

- Power-optimized design
- Programmable reduced frequency modes: fc/2, fc/4, fc/8 (where fc is the full-speed frequency of the processor)
- 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. This makes the TX19 suitable for interrupt-heavy applications in which immediate action is required at a higher priority level than the current processor priority level.
- Automatic update of the interrupt mask level

Processor Core for System ASIC Applications

- Unified manufacturing process and development environment
- 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 (μITRON) 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 is offered 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 know 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 cost of implementing numerous instructions made some designers think of streamlining the instruction set.

Feature 1

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, with Toshiba's 900/L1, the instruction "ADD A, (1000H)" brings the contents of memory location 1000H into the CPU, adds it to register A and places the results back 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 such 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 takes up a sizable amount of chip's real estate, creating a bottleneck in an effort to improve processor performance.

Feature 2

RISC processors have a fixed instruction size. In a CISC microprocessor, instructions can be 1, 2 or even 7 bytes. This variable instruction size makes the task of the instruction decoder very difficult since the size of the incoming instruction can never be known. In the TX19 microprocessor, the instruction size is fixed at 32 bits. The fixed instruction size enables the CPU to decode instructions quickly.

Feature 3

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 in which each instruction in a instruction pipeline can require a different number of clock cycles. Generally, RISC processors are heavily pipelined.

1.3 Features of the TX19

The previous section provided an overview of the features that make RISC processors set apart from CISC processors. In this section, we explore how the instruction set architecture (ISA) is implemented in the TX19. Where pertinent, comparisons are made with the 870/X and the 900/L1, 8-bit and 16-bit CISC processors from Toshiba.

The TX19 has two ISA modes, 16-bit and 32-bit. It provides for efficient run-time switching between 16-bit and 32-bit ISA modes through an instruction. The 16-bit instruction set (MIPS16) is not really a separate instruction set, but a 16-bit extension of the full 32-bit MIPS architecture. The 32-bit ISA has 85 instructions, the 16-bit ISA 53 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 TX19 did away with complex instructions**

  The TX19 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. The exceptions are the multiply-add instructions (MADD and MADDU) which require very fast processing. (These instructions are executed by the dedicated MAC circuitry.)

- **The TX19 did away with instructions that can be implemented by some other instructions**

  To reduce the size of the instruction set, the TX19 aggressively eliminated the instructions that can be implemented using other instructions. For example, the TX19 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 TX19 processors:

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

In the TX19, 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
```

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

```
ADDIU rt,rs,-1
```

- **The TX19 discarded instructions synthesizable from two or more simple instructions**

  The TX19 further pared down the instruction set by discarding the instructions that can be performed by two or more simple instructions. For example, the TX19 does not have the POP and 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 TX19, one of the 32 general-purpose registers is used as a stack pointer. The TX19 supports pushing onto the stack by executing an add instruction on the stack pointer and a store instruction.

- **The TX19 uses the load/store architecture**

  In the TX19, load and store instructions are the only instructions that move data between memory and CPU general registers. In such CISC processors as the 870/X and the 900/L1, data can be manipulated while it is still in memory. The TX19 did away with this kind of instructions like ADD, A, (1000H).

- **The TX19 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 order to simplify hardware implementation, in 32-bit ISA mode, the TX19 has only one addressing mode for accessing memory locations, i.e., based relative. In 16-bit ISA mode, the TX19 has two more addressing modes called PC-relative and SP-relative; only three 16-bit instructions can use these addressing modes, though.

- **The TX19 has three-operand computational instructions**

  In the TX19, many computational instructions use what is called 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*. Contrast this to

```
ADD XWA,XBC
```

for the 900/L1 which adds the contents of XWA and XBC and puts the result in XWA.

- **The TX19 does not have a flag register**

  The TX19 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 TX19 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 TX19, the result of compare instructions such as SLT (Set On Less Than) is placed into a general register.

1.3.2 Instruction Format

The TX19 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.

Each 16-bit instruction corresponds to exactly one 32-bit instruction. The 16-bit instructions are mapped to 32-bit instructions on the fly by relatively simple translation hardware. This is done serially as a preprocessor before the standard instruction decoder.

The size of the 870/X instructions are 1, 2, 3, 4, 5 or even 6 bytes. The 900/L1 has a 7-byte instruction. Although this variable instruction size is useful to reduce code size, it makes the task of the instruction decoder very difficult and slow since the size of the incoming instruction is never known.

1.3.3 Instruction Pipelines

The TX19 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 TX19 is that the execution of most instructions requires a uniform number of clock cycles; thus the TX19 is relatively easy to pipeline. The TX19 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 complex. Moreover, such a varied, complex instruction stream makes it almost impossible for a compiler to schedule instructions to reduce or eliminate pipeline stalls. For example, the instructions for the 870/X, which contains a 3-stage pipeline, takes 4 to 60 cycles to execute. The instructions for the 900/L1, also with a 3-stage pipeline, takes 2 to 27 cycles.

Chapter 2 CPU Architecture Overview

This chapter describes how data is represented in the CPU registers and in memory and also provides an overview of the functionality of the registers implemented in the TX19.

2.1 Data Formats

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

2.1.1 Byte Ordering

The TX19 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 TX19 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 TX19 components, byte ordering is fixed to either big-endian or little-endian.)

Figure 2-1shows the ordering of bytes in a word for the big-endian and little-endian formats. The TX19 processor uses byte addressing. Big-endian ordering assigns the lowest address to the highest-order (leftmost) byte. Little-endian ordering assigns the lowest address to the lowest-order (rightmost) byte. Notice that, in 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.

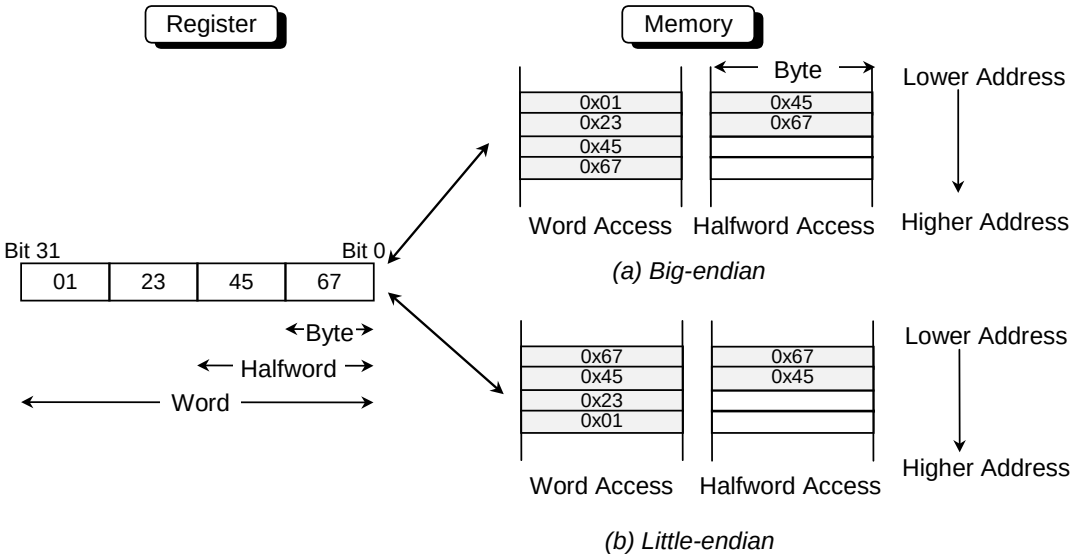


Figure 2-1 Byte Ordering

2.1.2 Aligned and Misaligned Accesses

The TX19 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. In other words, the natural address of an operand is an integer multiple of the operand length. A memory operand is said to be aligned if its address is a multiple of two for halfword accesses or a multiple of four for word accesses.

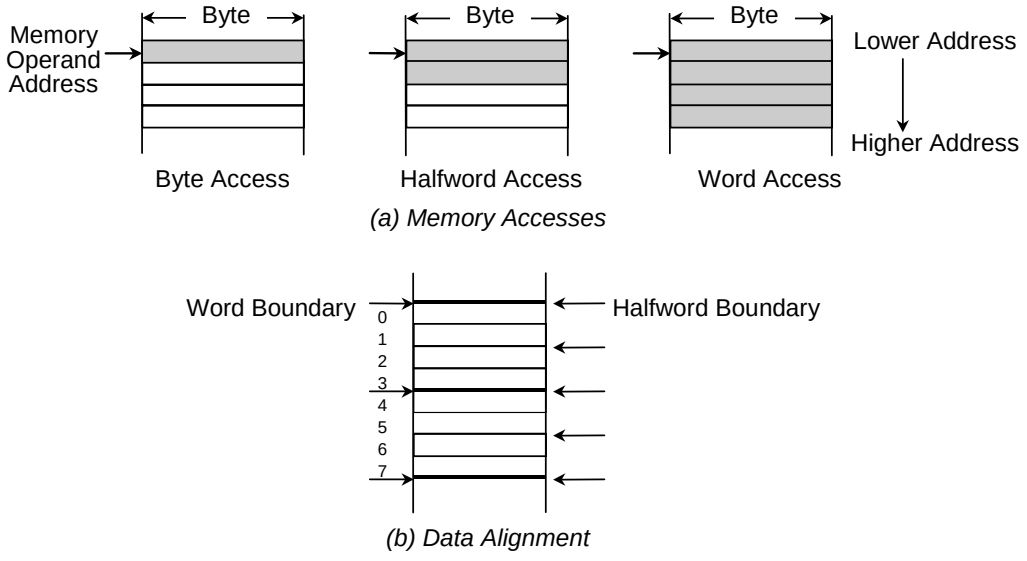


Figure 2-2 Aligned Data Items

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.

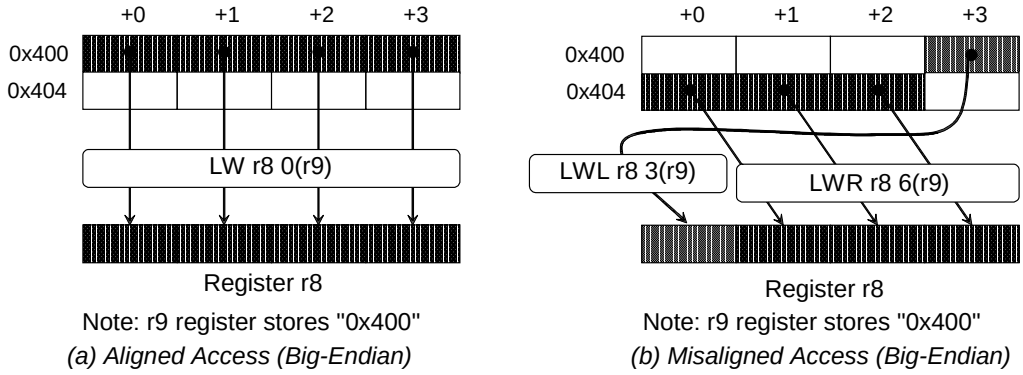
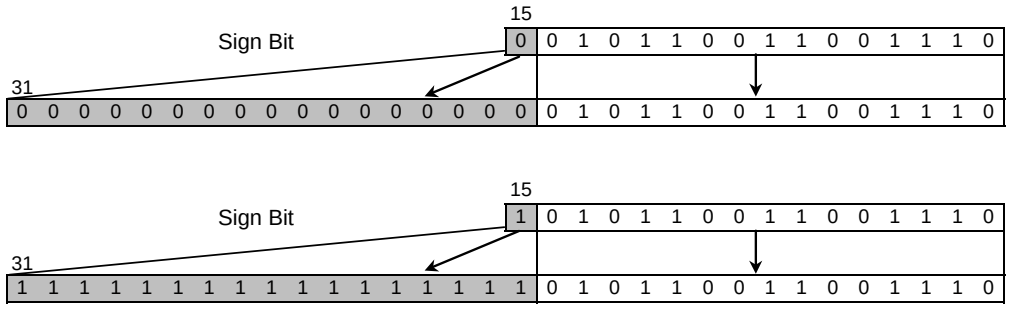


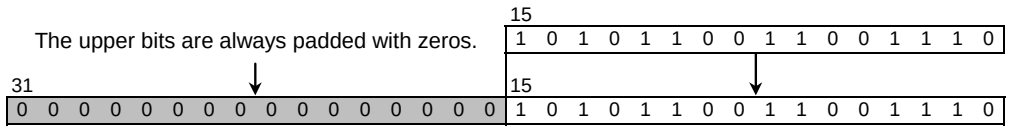
Figure 2-3 Aligned and Misaligned Accesses

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 the 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 the 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 can only 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 TX19 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), etc. since the only addressing mode supported is base register plus 16-bit immediate (i.e., offset). For example, the instruction "LW 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.

The LB (Load Byte) instruction, for example, treats the byte at the specified memory location as a signed number whereas the LBU (Load Byte Unsigned) assumes an unsigned number like the ASCII code of a character. Therefore, the LB instruction sign-extends the loaded byte and puts 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 on two source registers whereas the ANDI (AND Immediate) and ORI (OR Immediate) take a 16-bit immediate. ANDI and ORI zero-extends the 16-bit immediate and combine it with the contents of a general register in a bit-wise logical AND or OR operation.

2.2 Programming Model

The TX19 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 TX19 has 32 general-purpose registers, 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.

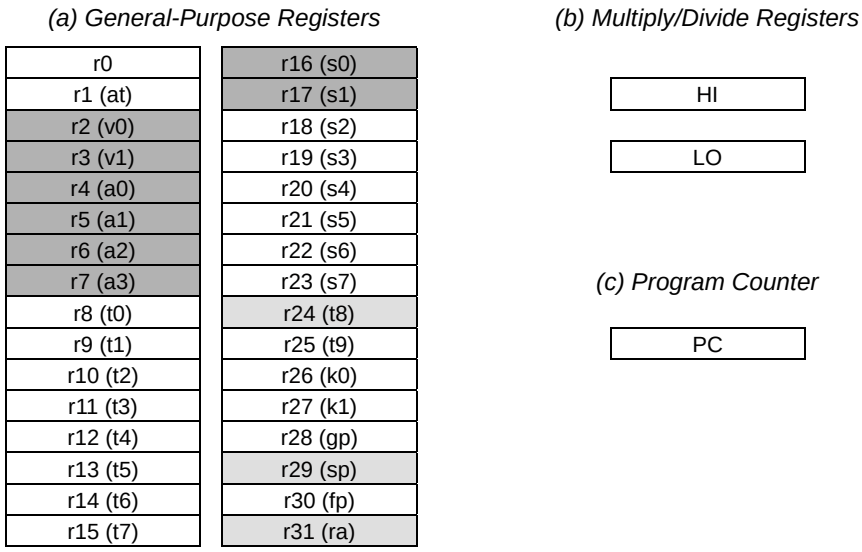


Figure 2-5 CPU Registers

General-Purpose Registers

The 32-bit ISA instructions can use any of the 32 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 at, v0-v1, a0-a3, and so on which 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 the link register used by Jump-and-Link, Branch-and-Link and Branch-Likely-and-Limk instructions. These instructions store an address at which processing resumes after a subroutine has been executed.

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, MIPS16 includes move instructions to copy values between the eight MIPS16 registers and the remaining 24 registers of the full MIPS architecture. Additionally, certain instructions can use r24 (t8), r29 (sp) and r31 (ra). r24 serves as a special condition register for handling compare results. r29 maintains the program stack pointer. r31 is the link register.

HI and LO Registers

The HI and LO registers hold the results of integer multiply, divide and multiply-add operations. Integer multiply and multiply-add 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 LO registers and general registers.

Program Counter (PC)

The least-significant bit of the program counter is the ISA mode bit that determines the width of instructions: 0 means 32-bit-wide instructions and 1 means 16-bit-wide instructions. This bit is not considered part of the address. The value formed by clearing it to 0 represents the address of the currently executing instruction.

2.2.2 System Control Coprocessor (CP0) Registers

The system control coprocessor, CP0, is an integral part of the TX19 processor. It has 32 registers of which nine registers shown in Figure 2-6 are accessible by users. These registers are all 32 bits in length.

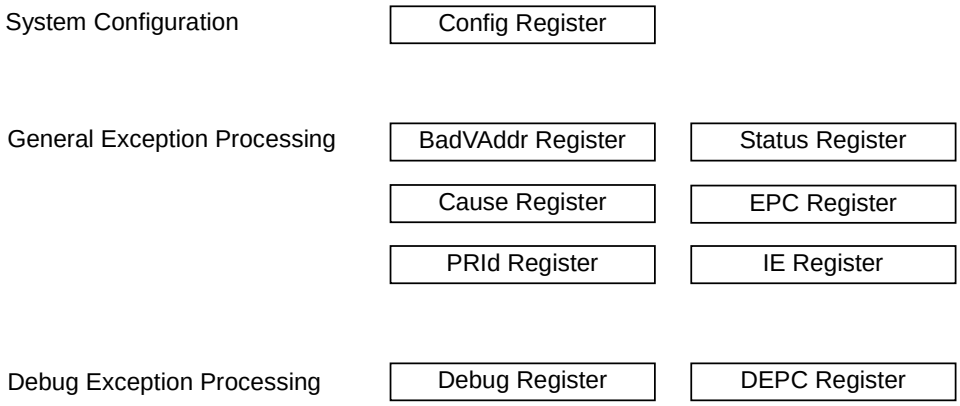


Figure 2-6 System Control Coprocessor (CP0) Registers

The CP0 registers are classified into three groups: system configuration register, 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 CU[0] bit in the Status register. When the processor is in User mode, the CP0 registers are accessible only when the CU[0] bit is 1. Operating modes are explained in Section 2.6, *Memory Management Summary.*

Table 2-1 System Configuration Register

Register Name	Description
Config	System configuration, e.g., power consumption management, cache enabling, etc.

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 enabling, etc.
Cause	Cause of the last exception
EPC	Exception program counter; i.e., address of the instruction that caused an exception
PRId	Processor revision identifier. Read-only
IE	Interrupt enable

Table 2-3 Debug Exception Handling Registers

Register Name	Description
Debug	Cause and current status of a debug exception
DEPC	Debug exception program counter; i.e., address of the instruction that caused a debug exception

2.3 32-Bit and 16-Bit ISA Modes

The TX19 has two ISA modes, 16-bit and 32-bit. It provides for efficient run-time switching between 16-bit and 32-bit ISA modes through an instruction. The TX19 supports whole procedures containing either 16-bit or 32-bit instructions, but it does not support mixing the two lengths together in a single procedure. 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-wide instructions and 1 means 16-bit-wide instructions. The JALX, JR or JALR instruction 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 and the ISA mode bit to the Exception Program Counter (EPC) or the Debug Exception Program Counter (DEPC). The JR instruction is used to jump back to the return address contained in the EPC 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 groupings shown in Figure 2-7.

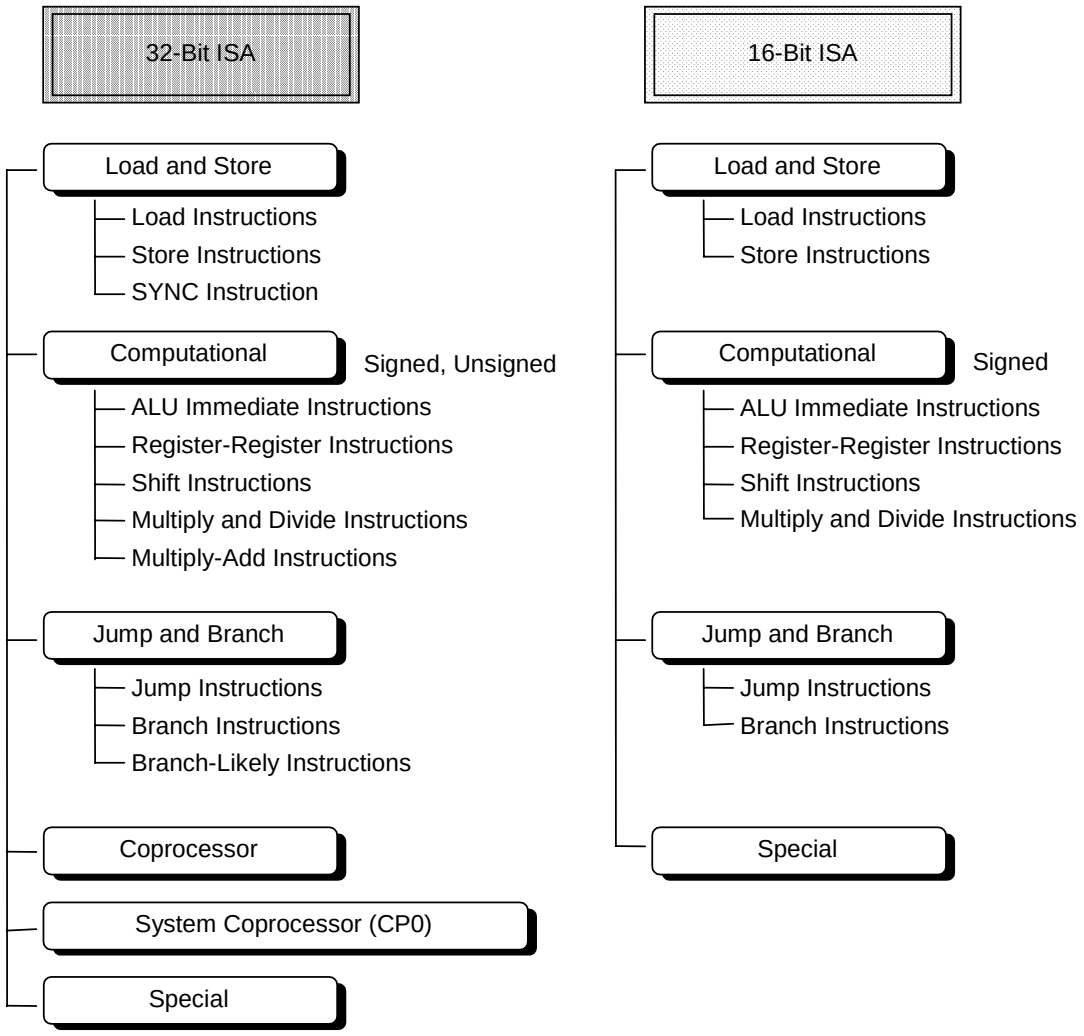


Figure 2-7 32-Bit and 16-Bit Instructions

All the instructions in the 32-bit ISA, as the name suggests, consist of 32 bits. All the instructions in the 16-bit ISA consist of 16 bits with the exception of JAL and JALX which are 32-bits wide. The EXTEND instruction for the 16-bit mode is 16-bits wide; it contains only an opcode and an immediate value. EXTEND does not generate a MIPS machine instruction on its own, but instead contributes 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.

Generally, each 16-bit instruction corresponds to exactly one 32-bit instruction. The 16-bit instructions fetched from main memory or an instruction cache are translated to 32-bit instructions on the fly by relatively simple translation hardware called decompressor. This is done serially as a preprocessor before the standard instruction decoder. Note that there are a few 16-bit instructions whose functions are slightly different from the 32-bit equivalents. Appendix B shows the mapping of the instruction format between 16-bit and 32-bit modes. Appendix B also provides supplemental remarks about instructions' functional differences, if any, between 16-bit and 32-bit modes.

2.4 Coprocessors

Coprocessors are secondary processors used to speed up operations by handling some of workload of the main CPU. The TX19 can operate with up to four coprocessors, CP0, CP1, CP2 and CP3.

CP0 is the system control coprocessor, which handles system configuration, exception handling and memory management. CP0 is an integral part of the TX19. The basic capabilities of CP0 is incorporated into the processor core and the extended capabilities into the memory management unit (MMU).

CP1, CP2 and CP3 are put outside of the processor core and are responsible for performing complicated and time-consuming tasks like floating-point mathematical functions. CP1, CP2 and CP3 are implementation-dependent; so they will be described in individual processor data sheets.

The CU[0] 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 CU[0] bit is cleared causes a Coprocessor Unusable exception. Kernel-mode programs can execute all CP0 instructions, regardless of the setting of the CU[0] bit.

The CU[3:1] bits in the Status register control accesses to the respective coprocessors whether 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 TX19 provides support for each of the four coprocessors to have up to 64 32-bit registers. The system control coprocessor (CP0) provides 32 registers of which nine registers are visible to the user. Chapter 8 gives a complete description of them.

2.5 Pipeline Architecture

The TX19 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 TX19 achieves an instruction execution rate approaching one instruction per clock cycle.

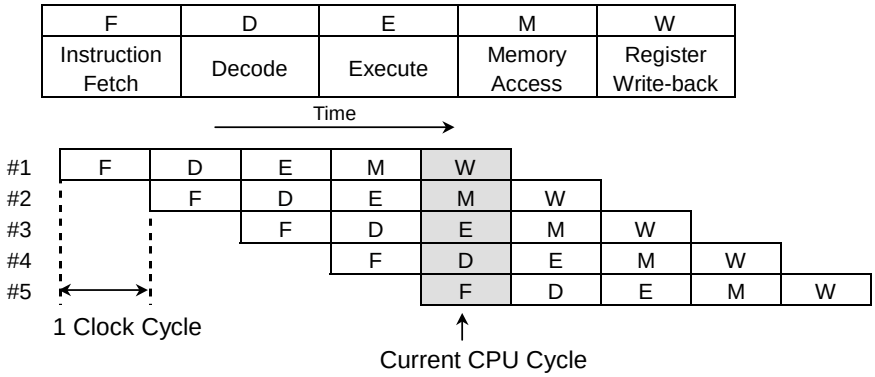


Figure 2-8 TX19 Pipeline

2.6 Memory Management Summary

The TX19 has two modes of operation, User mode and Kernel mode. The TX19 enters Kernel mode whenever an exception is taken. Since a reset exception occurs when a system is reset, the TX19 wakes up in Kernel mode. The processor switches to User mode when the RFE (Restore From Exception) or DERET (Debug Exception Return) instruction is executed.

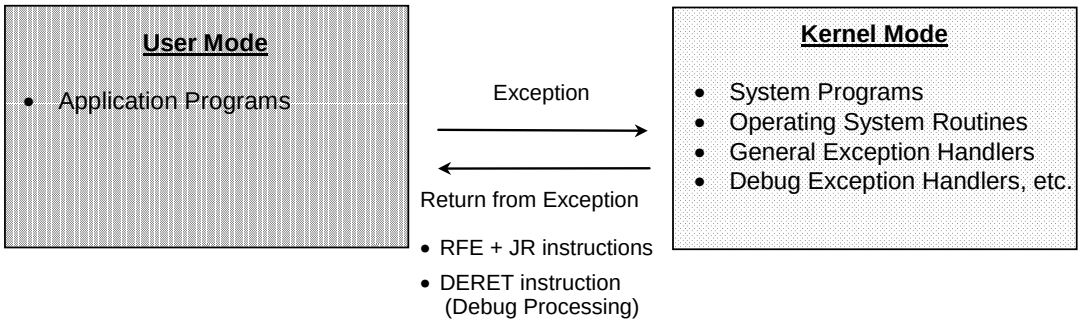


Figure 2-9 Operating Modes

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 is 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.

The TX19 does not contain a translation lookaside buffer (TLB). Instead, the memory management unit (MMU) of the TX19 uses the direct segment mapping method. The mapping of virtual addresses to physical addresses is shown in Figure 2-10. 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 MMU in greater details.

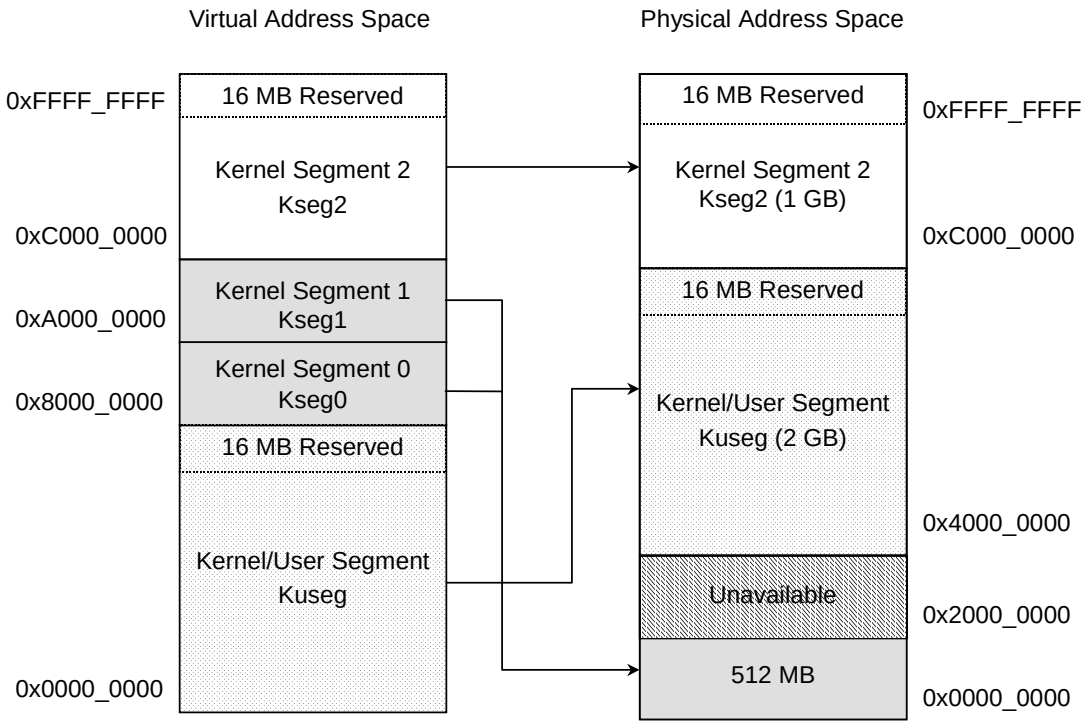


Figure 2-10 Virtual-to-Physical Address Mapping

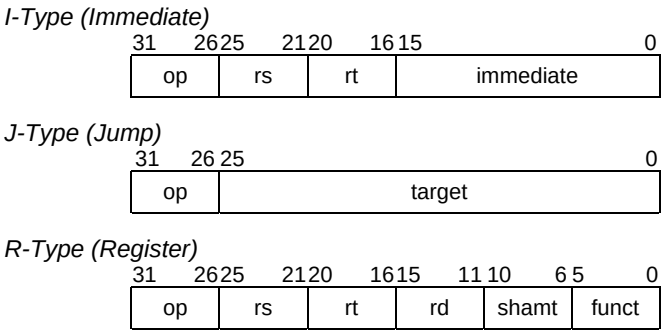
Chapter 3 32-Bit ISA Summary and Programming Tips

This chapter gives an overview of the instructions and addressing modes supported by the TX19 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
- Coprocessor instructions
- Special instructions

3.1 Instruction Formats

All TX19 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 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 can be used to implement immediate addressing using r0 as the base register or register direct addressing using an offset value of zero.

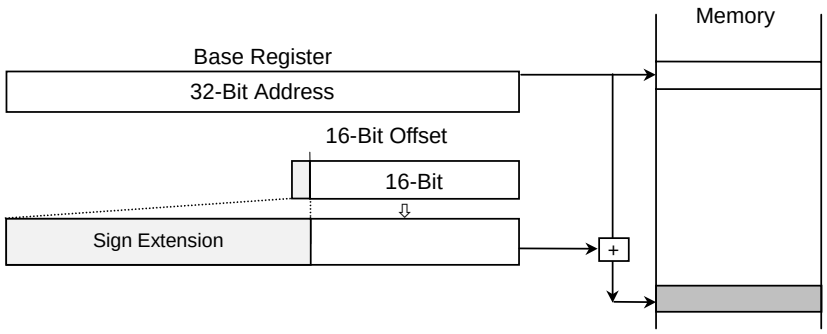


Figure 3-2 Register Indirect with Offset Addressing

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 attempt is made to load or store halfword or word that is not aligned on the natural alignment boundary. 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 combination. Likewise, the SWL (Store Word Left) and SWR (Store Word Right) instructions are used in combination. These instructions provide a more efficient way of dealing with misaligned data than is possible using a sequence of load/store and shift operations. They are useful for compatibility with 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, stores or cache refills after this instruction are allowed to start. See Chapter 5, *CPU Pipeline*, for more on this.

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. Setting aside the most-significant bit for the sign leaves a total of 15 bits for 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 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.

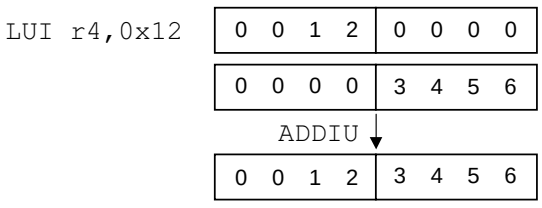
```
ADDU      r4,r4,r5
LW        r6,0(r4)
```

- 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) instructions 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 an 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 will observe how the integer overflow is trapped using software routines. In Section 3.3.5, we will look at ways to execute a 64-bit x 64-bit multiply operation. The 32-bit ISA has no rotate instructions; 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 three register operands.

Table 3-3 Computational Instructions

Category	Instruction	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
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
Shift	Logical Shift	SLL, SLLV, SRL, SRLV
	Arithmetic Shift	SRA, SRAV
Multiply and Divide	Multiply	MULT, MULTU
	Divide	DIV, DIVU
	Move From/To HI/LO	MFHI, MFLO, MTHI, MTLO
Multiply-and-Add		MADD, MADDU

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 and places the result into the destination register (*rd*).

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, SRAV) do not have the shift amount (*shamt*) field; instead they specify a general-purpose register containing the desired shift amount.

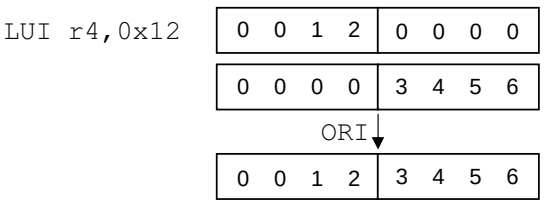
Multiply and divide instructions operate on integer values in two general-purpose registers and place the result into special registers HI and LO. Generally, CPU instructions do not have access to the HI and LO registers. In the MIPS architecture, the MFHI, MFLO, MTHI and MTLO instructions are always required to move data between a general-purpose register and the HI or LO register. However, the TX19 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 instructions are extended instructions implemented in the TX19. They 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 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 gives 0x12345678, which 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.

Note: Typically, the assembler accepts immediate values longer than 16 bits. For example, when you write this instruction:

```
ADDI r3,r2,0x12345678
```

the assembler automatically breaks it into a sequence of multiple instructions, as shown below:

```
LUI r1,0x1234
ORI r1,r1,0x5678
ADD r3,r2,r1
```

This assembler capability eases your programming. As demonstrated by this example, register r1 is reserved for use by the assembler. Don't use it in your assembly-language program.

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.

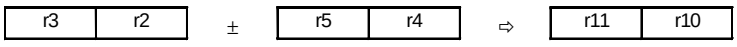


Figure 3-3 64-Bit Addition and Subtraction

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 TX19 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. For the sake of discussion, let's assume that the two operands are positive values. Then, based on the fact that if the sum is less than one of the operands added, a carry-out occurred, 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, trap (i.e., 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 traps 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 trap
XOR  r5,r3,r4   #  Compare signs of r3 and r4; if different,
                #  overflow never occurs (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, r5 => 0
                    #  (overflow never occurs)
    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 (r5 < 0)
    BLTZ r5,Ov      #  Branch on less than zero
No_Ov:
```

3.3.5 64-Bit x 64-Bit Multiplication

To multiply two integer numbers in the TX19, they must be in general-purpose registers. In doubleword-by-doubleword multiplication, each 64-bit operand take 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.

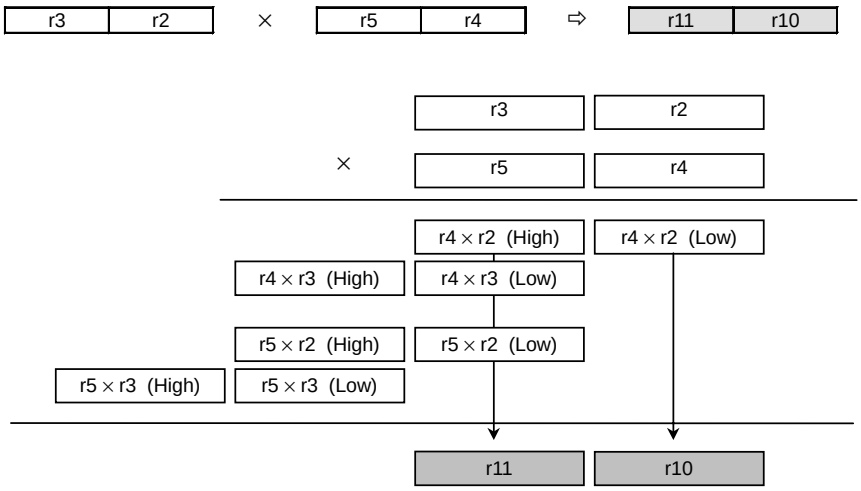


Figure 3-4 64-Bit x 64-Bit Multiplication

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 r2, 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 TX19 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 TX19, however, the MULTU instruction can take a third operand. In the TX19, 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 TX19, there are no rotate instructions at the machine level (although assemblers may have macro instructions that perform rotate left and rotate right). In rotate left, for example, as bits are shifted from right to left, they exit from the left end (MSB) and enter the right end (LSB). In shift left, bits that exit the left end are discarded and zeros are supplied to the vacated bits on the right.

In the TX19, a rotate operation must be implemented using shift and logical-OR instructions. Figure 3-5 illustrates how to do this.

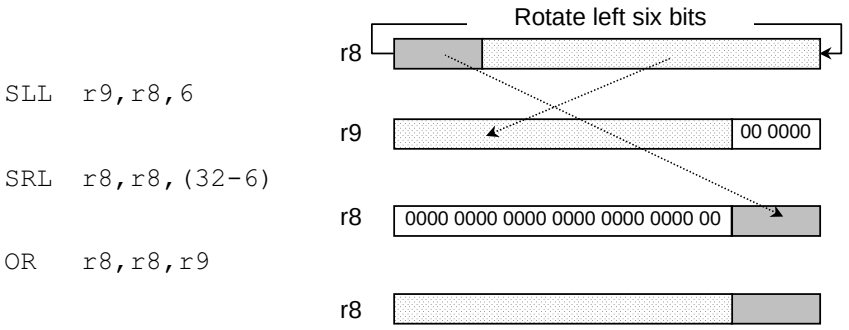


Figure 3-5 Rotate Left by 6 Bits

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 TX19 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 3.4.5,

the differences between regular branch instructions and branch-likely instructions are explained. Section 3.4.6 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 TX19, 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
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 all the jump and regular branch, 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.* Branch-likely instructions are detailed in Section 3.4.4.

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. As shown in Figure 3-6, 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.

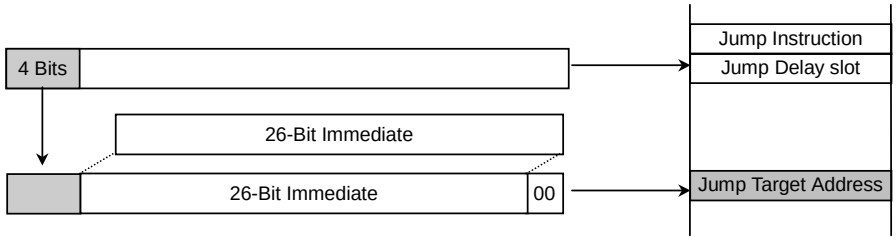


Figure 3-6 Paged Absolute Addressing (32-Bit ISA Mode)

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 whose two least-significant bits are zero.

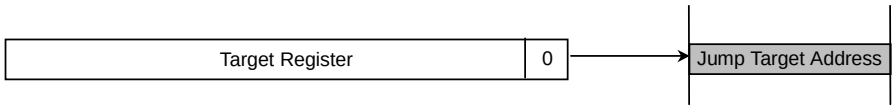


Figure 3-7 Register Indirect Addressing (32-Bit ISA Mode)

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 using PC-relative with offset addressing. As shown in Figure 3-8, 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.

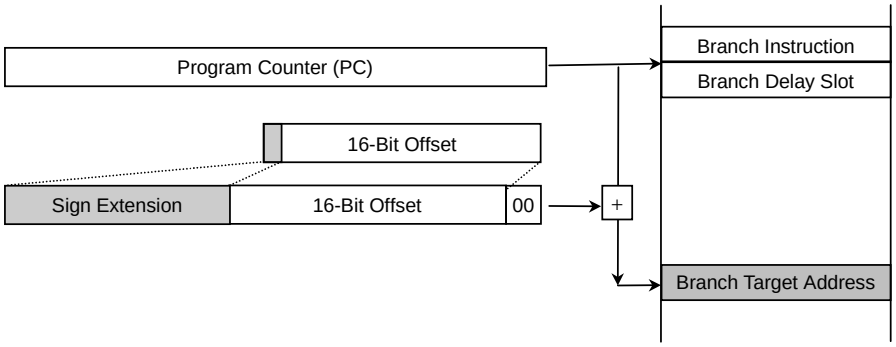


Figure 3-8 PC-Relative with Offset Addressing (32-Bit ISA Mode)

3.4.3 Run-Time Switching of the ISA Modes

The TX19 has two ISA modes, 16-bit ISA and 32-bit ISA. The TX19 provides for efficient run-time 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 whose two least-significant bits are zero. If these bits are not zero, 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.

3.4.4 Branch-Likely Instructions

All the jump and branch instructions occur with a delay of two instructions 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 TX19 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 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.

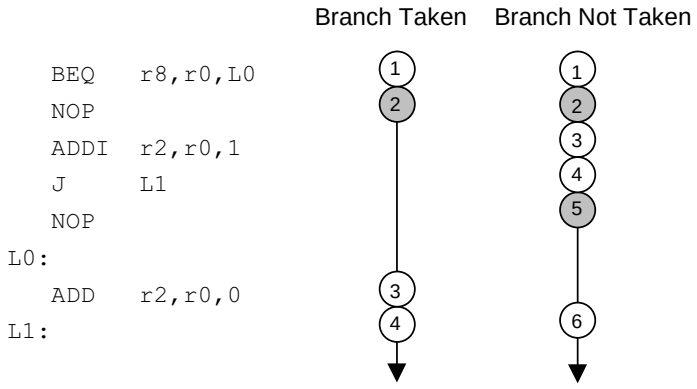


Figure 3-9 Regular Branch Instruction

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.

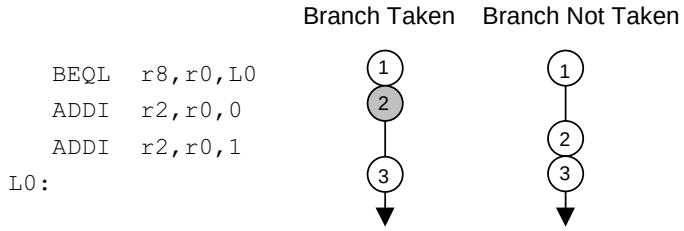


Figure 3-10 Branch-Likely Instruction

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 only 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. (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 ≥ 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 r24 to 1. Otherwise, r24 is set to 0. The BEQ instruction branches to Label if r24 is 0 (Remember r0 is hardwired to a constant value of zero).

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

- Example 2: Branch if r7 ≥ 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 checks if the contents of r10 is equal to 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 2^{28}-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 unconditionally 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.

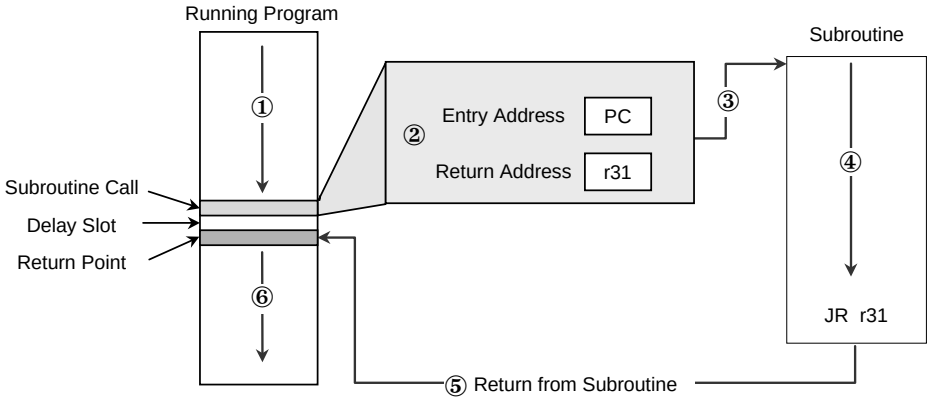


Figure 3-11 Subroutine Calls and Returns

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 jump, branch or branch-likely instruction that precedes it is set to the Exception Program Counter (EPC) register. After the exception handler routine has been executed, processing restarts with the jump, branch or branch-likely instruction and the instruction in the delay slot. Because jump, branch and branch-likely instructions can be restarted after exceptions or interrupts, 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 TX19 can operate with up to four coprocessors, CP0, CP1, CP2 and CP3. Instructions categorized under coprocessor instructions perform operations on CP1 to CP3. Coprocessor load and store instructions are I-type. Coprocessor computational instructions have coprocessor-dependent formats.

The CU[3:1] bits in the Status register control accesses to the respective coprocessors whether in User mode or in Kernel mode. Attempted execution of a coprocessor instruction causes a Coprocessor Unusable exception when its CU bit is cleared.

Table 3-6 shows the coprocessor instructions other than CP0 instructions (where *z* is the coprocessor number).

Table 3-6 Coprocessor Instructions (32-Bit ISA)

Name	Opcode
Move To/From Coprocessor	MTC*z*, MFC*z*
Move Control To/From Coprocessor	CTC*z*, CFC*z*
Coprocessor Operation	COP*z*
Branch on Coprocessor *z* True/False	BC*z*T, BC*z*F
Branch on Coprocessor *z* True/False Likely	BC*z*TL, BC*z*FL

The Load Word To Coprocessor (LWC*z*) and Store Word From Coprocessor (SWC*z*) instructions available with the MIPS R3000A are not supported by the TX19. 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 CU[0] 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 CU[0] bit is cleared causes a Coprocessor Unusable exception. Kernel-mode programs can execute all CP0 instructions, regardless of the setting of the CU[0] bit. Table 3-7 shows the CP0 instructions.

Table 3-7 System Control Coprocessor (CP0) Instructions

Name	Opcode
Move To/From CP0	MTC0, MFC0
Restore From Exception	RFE
Debug Exception Return	DERET
Cache Operation	CACHE

The TX19 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 traps, i.e., to test for a particular condition in a running program. All special instructions are R-type. The 32-bit ISA has three special instructions, SYSCALL (System Call), BREAK (Breakpoint) and SDBBP (Software Debug Breakpoint). SDBBP is an extension implemented in the TX19; it is not part of the MIPS R3000A architecture. Special instructions transfer program control to an appropriate exception handler. For details on exception processing, see Chapter 6.

3.7 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 implemented in the TX19 that are not part of the TX39 or MIPS R3000A architecture. For a complete list of differences in the instruction set between the TX19, the TX39 and the MIPS R3000A, 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 signed-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 signed-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.

Instruction	Format	Operation
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	**This instruction is an extension to the R3000A architecture.** 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. Traps 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 trap 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 Three-Operand Register-Type Instructions (32-Bit ISA)

Instruction	Format	Operation
Add	ADD *rd*, *rs*, *rt*	The sum *rs* + *rt* is placed into *rd*. Traps on 2's-complement overflow.
Add Unsigned	ADDU *rd*, *rs*, *rt*	The sum *rs* + *rt* is placed into *rd*. Does not trap on 2's-complement overflow.
Subtract	SUB *rd*, *rs*, *rt*	The remainder *rs* - *rt* is placed into *rd*. Traps on 2's-complement overflow.
Subtract Unsigned	SUBU *rd*, *rs*, *rt*	The remainder *rs* - *rt* is placed into *rd*. Does not trap on 2's-complement 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-OR	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*.

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 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 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 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	MULT (*rd*,) *rs*, *rt*	**The *rd* operand is an extension to the R3000A architecture.** 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 *rd* operand is an extension to the R3000A architecture.** 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.

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

Instruction	Format	Operation
Multiply-and-Add	MADD (*rd*,) *rs*, *rt*	**This instruction is an extension to the R3000A architecture.** 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*	**This instruction is an extension to the R3000A architecture.** 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*.

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

Instruction	Format	Operation
Jump	J *target*	A jump is taken 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*	A jump is taken 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*	**This instruction is an extension to the TX39 and R3000A architectures.** A jump is taken 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*	A jump is taken to the address specified by the upper 31 bits of *rs*. The least-significant bit of *rs* is interpreted as the ISA mode specifier.
Jump And Link Register	JALR (*rd*,) *rs*	A jump is taken to the address specified by the upper 31 bits of *rs*. 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*	**BEQL is an extension to the R3000A architecture.** If *rs* = *rt*, a branch is taken 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*	**BNEL is an extension to the R3000A architecture.** If *rs* ≠ *rt*, a branch is taken 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*	**BGTZL is an extension to the R3000A architecture.** If *rs* > 0, a branch is taken 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*	**BGEZL is an extension to the R3000A architecture.** If *rs* ≥ 0, a branch is taken 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*	**BLTZL is an extension to the R3000A architecture.** If *rs* < 0, a branch is taken 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*	**BLEZL is an extension to the R3000A architecture.** If *rs* ≤ 0, a branch is taken 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*	**BLTZALL is an extension to the R3000A architecture.** If *rs* < 0, a branch is taken 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*	**BGEZALL is an extension to the R3000A architecture.** If *rs* ≥ 0, a branch is taken 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.

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

Table 3-16 Coprocessor Instructions (32-Bit ISA)

Instruction	Format	Operation
Move To Coprocessor	MTC*z* *rt*, *rd*	The contents of general register *rd* is copied into coprocessor register *rt* of coprocessor unit *z*.
Move From Coprocessor	MFC*z* *rt*, *rd*	The contents of coprocessor register *rd* of coprocessor unit *z* is copied into general register *rt*.
Move Control To Coprocessor	CTC*z* *rt*, *rd*	The contents of general register *rt* is copied into coprocessor control register *rd* of coprocessor unit *z*.
Move Control From Coprocessor	CFC*z* *rt*, *rd*	The contents of coprocessor control register *rd* of coprocessor unit z is copied into general register *rt*.
Coprocessor Operation	COP*z* *cofun*	Coprocessor unit *z* performs the operation specified by *cofun*.
Branch On Coprocessor *z* True (Likely)	BC*z*T(L) *offset*	If the coprocessor unit *z* condition line is true, a branch is taken 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 Coprocessor *z* False (Likely)	BC*z*F(L) *offset*	If the coprocessor unit *z* condition line is false, a branch is taken to the target address specified as a 16-bit *offset* relative to PC + 4 (i.e., the address of the branch delay slot).

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

Table 3-17 System Control Coprocessor (CP0) Instructions (32-Bit ISA)

Instruction	Format	Operation
Move To CP0	MTC0 *rt*, *rd*	**This is an extension to the R3000A architecture.** The contents of general register *rt* is copied into CP0 register *rd*.
Move From CP0	MFC0 *rt*, *rd*	**This is an extension to the R3000A architecture.** The contents of CP0 register *rt* is copied into general register *rd*.
Restore From Exception	RFE	**This is an extension to the R3000A architecture.** The old status bits (interrupt enable and operating mode) of the Status register are restored into the previous status bits, and the previous status bits are restored into the current status bits. Additionally, the previous interrupt mask level field is restored to the current mask level field.
Debug Exception Return	DERET	**This is an extension to the R3000A architecture.** 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.
Cache	CACHE *op*, *offset*(*base*)	**This is an extension to the R3000A architecture.** A virtual address is formed by adding *offset* and *base* and this virtual address is translated into a physical address. *op* specifies a cache operation for this address.

Table 3-18 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 trap occurs, immediately and unconditionally transferring control to the exception handler.
Software Debug Breakpoint Exception	SDBBP *code*	**This is an extension to the R3000A architecture.** A debug breakpoint trap occurs, immediately and unconditionally transferring control to the exception handler.

Chapter 4 16-Bit ISA Summary and Programming Tips

This chapter gives an overview of the instructions and addressing modes supported by the TX19 in 16-bit ISA mode. This chapter also presents many programming tips using 16-bit instructions. Instructions are grouped into the following categories. Branch-likely and coprocessor instructions are not supported by the 16-bit ISA.

- Load and store instructions
- Computational instructions
- Jump and branch instructions
- Special instructions

Doubleword instructions available in the MIPS16 ASE are not implemented in the TX19.

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 MOVE instructions to copy values between the eight 16-bit-ISA registers and the remaining 24 registers of the full 32-bit architecture. Additionally, certain instructions implicitly use r24 (t8), r29 (sp) and r31 (ra). r24 serves as a special condition register for handling compare results. r29 maintains the program stack pointer. r31 is the link register. Multiply and divide instructions use the special registers HI and LO.

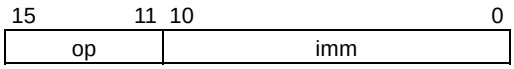
4.1 Instruction Formats

The TX19 instructions for the 16-bit ISA mode are all 16-bits wide, except JAL and JALX which are 32-bits wide. Basically, there are ten instruction formats for 16-bit instructions as shown in Figure 4-1. The 32-bit JAL and JALX instructions use the JAL/JALX format shown in Figure 4-2.

To fit within the 16-bit limit, immediate fields in the 16-bit instructions are only 4 to 11 bits. 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 2- to 8-bit 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, as shown in Figure 4-2. For example, the EXTENDed version of the I-type format is called EXT-I.

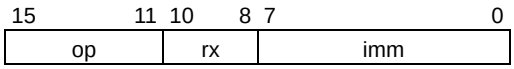
<< 16-Bit Instructions >>

I Type



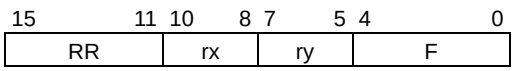
(op: B)

RI Type

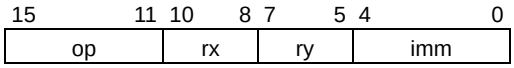


(op: ADDIU8, ADDIUPC, ADDIUSP, BEQZ, BNEZ, CMPI, LI, LWPC, LWSP, SLTI, SLTIU SWSP)

RR Type

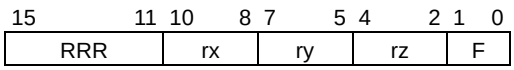


RRI Type

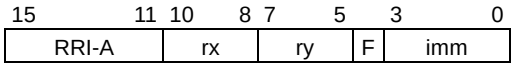


(op: LB, LBU, LH, LHU, LW, SB, SH, SW)

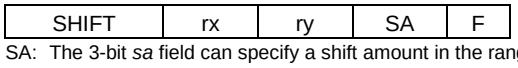
RRR Type



RRI-A Type

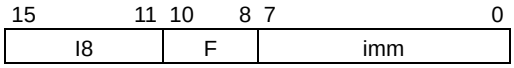


SHIFT Type

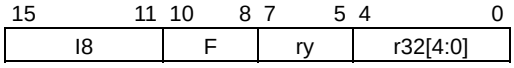


SA: The 3-bit *sa* field can specify a shift amount in the range of 1 to 8. The 16-bit ISA defines the value 0 in the *sa* field to mean a shift of 8 bits.

I8 Type

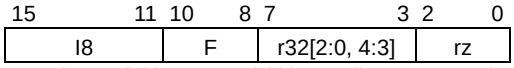


I8_MOVR32 Type



F: BTEQZ, BTNEZ, SWRASP, ADJSP, MOV32R, MOVR32

I8_MOV32R Type



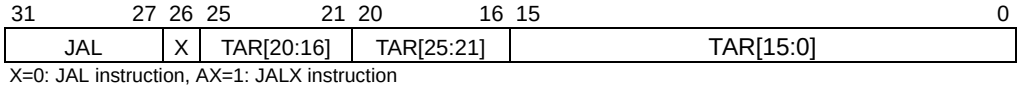
r32: The *r32* field uses special bit encoding. For example, encoding of register r7 (00111) is 11100 in the *r32* field.

op	5-bit operation code
rx	3-bit source/destination register specifier
ry	3-bit source/destination register specifier
immediate or imm	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

Figure 4-1 16-Bit Instruction Formats

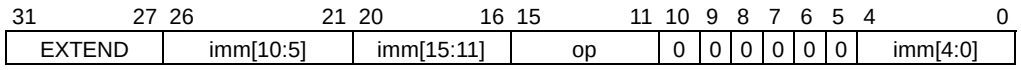
<< 32-Bit Instructions >>

JAL and JALX Type

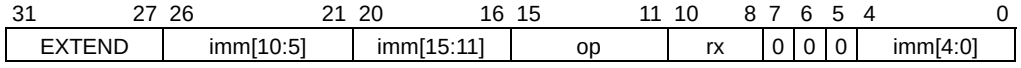


X=0: JAL instruction, AX=1: JALX instruction

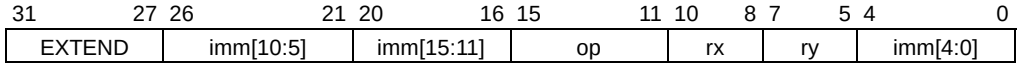
EXT-I Type



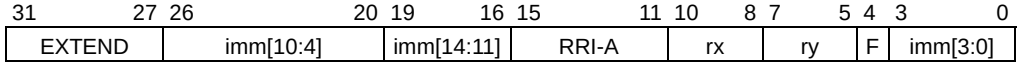
EXT-RI Type



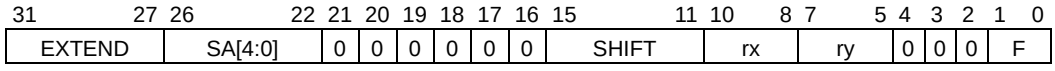
EXT-RRI Type



EXT-RRI-A Type



EXT-SHIFT Type



EXT-I8 Type

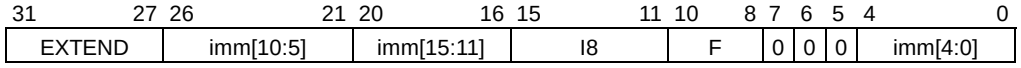


Figure 4-2 32-Bit Instruction Formats

4.2 Load and Store Instructions

In the 16-bit ISA, there are no load/store instructions for misaligned data and the SYNC instruction for memory synchronization. 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 or 8 bits of unsigned values. To overcome this restriction, the 16-bit ISA contains a mechanism to EXTEND instructions with 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 a new addressing mode.

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 a new addressing mode.

4.2.1 Load and Store Address Calculation

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

- Register indirect with offset
- SP-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 shifted by two bits. In the case of halfword accesses, the offset is shifted by one bit.

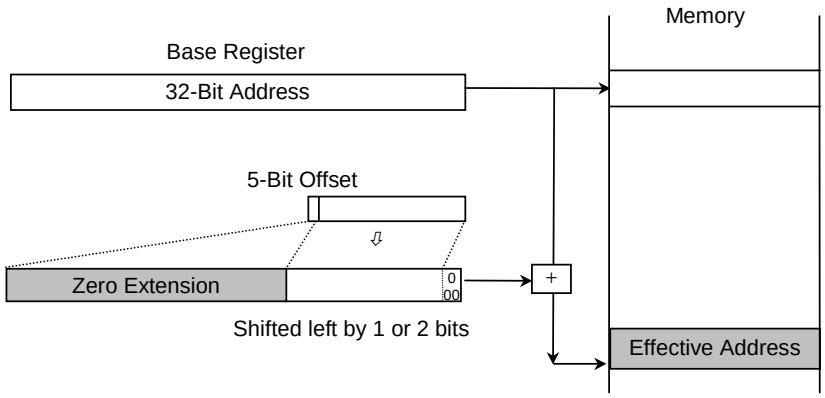


Figure 4-3 Register Indirect with Offset Addressing (16-Bit ISA)

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 special stack pointer and is called sp. The 16-bit ISA refers to it implicitly through special function codes, 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 Load Word (LW) and Store Word (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.

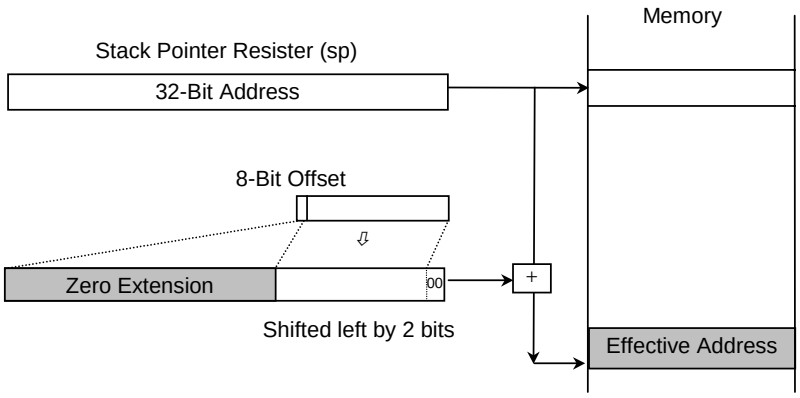


Figure 4-4 SP-Relative Addressing (16-Bit ISA)

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 and adding the resultamt 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.

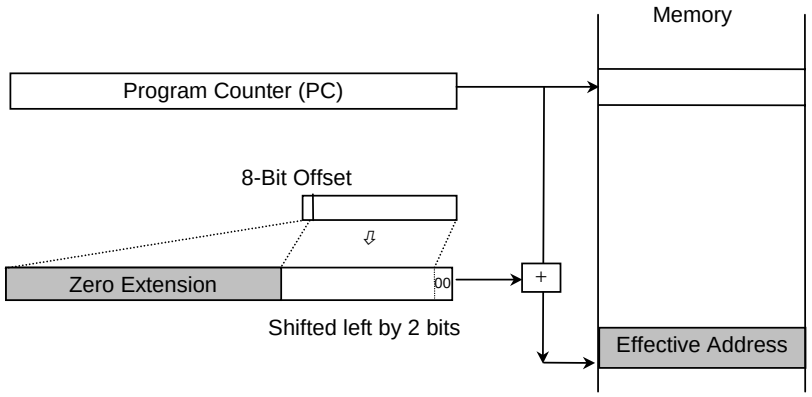


Figure 4-5 PC-Relative with Offset Addressing (16-Bit ISA)

4.2.2 Overview of Load and Store Instructions

Table 4-1 and Table 4-2 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.

Byte and halfword loads and stores use register indirect with offset addressing. Word loads and stores support all the addressing modes described in the previous section, with one exception that SW does not support PC-relative addressing.

Table 4-1 Load Instructions

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

Table 4-2 Store Instructions

Data Type	Opcode	Addressing
Byte	SB	Register Indirect
Halfword	SH	Register Indirect
Word	SW	Register Indirect SP-relative

4.2.3 32-Bit Address Generation

In 16-bit ISA mode, the offset field is restricted to only 5 or 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 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 an addition of an immediate value into the lower 16 bits. However, the 16-bit ISA does not have the LUI instruction. Instead, the 16-bit ISA has PC-relative addressing. In the example below, the memory location addressed by the first LW instruction contains a 32-bit offset value. It loads the offset value into r5. The ADDU instruction then adds it to the base address held in r4 to form the effective address. This way, the last LW instruction can use r4 to address the desired memory location, with an offset of zero.

```
LW      r5,16(pc)
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. The second LW instruction can address a desired memory location, with an offset of zero.

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

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 differences between the 16-bit ISA and the 32-bit ISA. 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 4-3. They consist of arithmetic, compare, logical, shift, multiply and divide. Multiply-and-add 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 4-3 Computational Instructions

Category	Instruction	Opcode
ALU Immediate	Add	ADDIU
	Set On Less Than	SLTI, SLTIU
	Compare	CMPI
	Load Immediate	LI
Register-Type	Add	ADDU
	Subtract	SUBU
	Set On Less Than	SLT, SLTU
	Compare	CMP
	Negate	NEG
	Logical AND	AND
	Logical OR	OR
	Logical XOR	XOR
	Not	NOT
	Move	MOVE
Shift	Logical Shift	SLL, SLLV, SRL, SRLV
	Arithmetic Shift	SRA, SRAV
Multiply and Divide	Multiply	MULT, MULTU
	Divide	DIV, DIVU
	Move From/To HI/LO	MFHI, MFLO

In ALU immediate instructions, the source operands are a general-purpose register and a 5- or 8-bit immediate. The 16-bit ISA did away with immediate logical instructions such as ANDI, ORI and XORI. However, the 16-bit ISA has a new instruction, CMPI, for compare operations; it exclusive-ORs the contents of a general register (*rs*) with the zero-extended immediate and puts the result in register t8 (r24). The LI instruction loads a register with the zero-extended immediate.

Except for the ADDIU instruction, the 5- and 8-bit immediates in the ALU immediate instructions are zero-extended. However, when EXTEND is prepended to these instructions, they use the conventional signed 16-bit immediate of 32-bit ISA mode.

Register-type instructions manipulate the values held in two general-purpose registers and place the result into a general-purpose register. There are two-operand (RR-type) and three-operand (RRR-type) instructions. The 16-bit ISA dropped arithmetic instructions that can trap in order to save opcode space. Instead, 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 visible to the 16-bit ISA and the remaining 24 registers of the full 32-bit architecture.

Load Immediate (LI), Negate (NEG) and Not (NOT) were added to the 16-bit ISA since these operations could no longer 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 16-bit ISA, however, 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 extends the 3-bit *sa* fields into 5-bit fields for shifts.

Multiply and divide instructions in the 16-bit ISA perform the same functions as those in the 32-bit ISA, except that, in the 16-bit ISA, MULT and MULTU do not have an extension to place the lower 32 bits of the product into a general-purpose register. In addition, with the multiply-and-add instructions gone, the Move To HI/LO instructions (MTHI, MTLO) were left out.

4.3.2 32-Bit Constants

Even EXTEND can extend immediate fields in computational instructions to only 16 bits. For immediates greater than 16 bits, you can not use the sequence of the LUI and ORI instructions as in 32-bit ISA mode because there is neither the LUI nor ORI instruction in the 16-bit ISA. Instead, in 16-bit ISA mode, 32-bit constants can be embedded in the code segment, typically between subroutine bodies. 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 the two 32-bit instructions required by the 32-bit ISA.

The following is an example of adding a 32-bit constant to the contents of a general register. The LW instruction loads a 32-bit constant into r5 from memory. The ADDU instruction adds the contents of r4 and r5 together and puts the result 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 LI (Load Immediate) instruction as follows:

```
LI rx,0
```

which zero-extends and loads the immediate value (0) into *rx*.

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 discarded all branch instructions that compare two registers and then branch, such as BEQ, BNE, BGEZ, BGTZ, BLEZ and BLTZ. To compensate for the loss of these instructions, the 16-bit ISA included 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 has branch instructions to test t8 and branch based on the zero or non-zero state of t8. The 16-bit ISA did away with branch-and-link instructions.

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 4-4 and Table 4-5 show the opcodes of the jump and branch instructions in the 16-bit ISA.

Table 4-4 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
JALR	Jump And Link Register	Register Indirect

Table 4-5 Branch Instructions (16-Bit ISA)

Opcode	Name	Condition	Addressing
BEQZ	Branch On Equal to Zero	*rx* = 0	PC-relative
BNEZ	Branch On Not Equal To Zero	*rx* ≠ 0	PC-relative
BTEQZ	Branch On T8 Equal To Zero	t8 = 0	PC-relative
BTNEZ	Branch On T8 Not Equal To Zero	t8 ≠ 0	PC-relative
B	Branch Unconditional	—	PC-relative

Jump-and-link 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 instead of by two bits.

Although the B instruction is an unconditional branch, it is grouped under the branch instruction category, not the jump. This is because the B instruction is translated into a 32-bit BEQ instruction comparing r0 and r0.

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.

Jumps still have a two-slot delay in the 16-bit ISA mode as in the 32-bit ISA mode.

Run-Time Switching of the ISA Modes

As shown in Table 4-1, the 16-bit ISA includes the JALX, JR and JALR instructions as in the 32-bit ISA. These instructions can still 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 only jump-and-link instructions (JALX, JALR). There are no branch-and-link or branch-and-link-likely instructions. 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
```

However, the 16-bit ISA has no logical immediate instructions like ORI and no access to r0. To compensate for this, the 16-bit ISA provides a new instruction, CMPI, to compare a register and an immediate and set 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 is 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

In the 16-bit ISA, the sequence of LUI and ORI can not create a 32-bit address due to the loss of the these instructions. However, in 16-bit ISA mode, 32-bit constants can be included in code. Given the new addressing mode, PC-relative, the LW instruction can be used to load a 32-bit constant from memory. For example:

```
LW      r4,0(pc)
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 Special Instructions

Special instructions include the BREAK (Breakpoint) and SDBBP (Software Debug Breakpoint) instructions. There is not the SYSCALL (System Call) instruction in the 16-bit ISA.

Additionally, the 16-bit ISA has a new instruction called EXTEND. EXTEND is not really an instruction that generates a machine instruction on its own. EXTEND provides a way to extend a short immediate in a 16-bit instruction to the full 16 bits. EXTEND consists of a 5-bit opcode and a 11-bit immediate field. Prepended to a 16-bit instruction with an immediate, EXTEND contributes its immediate to be merged with the short immediate in the following instruction. Table 4-6 shows the length of the immediate field in instructions before and after they are EXTENDed.

Table 4-6 EXTENDable Instructions

16-Bit Instruction		Immediate Bit Size	
		Before EXTENDed	After EXTENDed
Load/Store	LB, LBU	5	16
	LH, LHU	5	16
	LW	5 (or 8)	16
	SB	5	16
	SH	5	16
	SW	5 (or 8)	16
Computational	ADDIU	4	15
		8	16
	SLTI, SLTIU	8	16
	CMPI	8	16
	LI	8	16
	SLL	3	5
	SRL	3	5
	SRA	3	5
Branch	BEQZ	8	16
	BNEZ	8	16
	BTEQZ	8	16
	BTNEZ	8	16
	B	11	16

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; the outcome of doing otherwise 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, ADDIU is an RI (register-immediate) type instruction, with a 8-bit immediate field. Therefore, the instruction:

```
ADDIU r3,0x1234
```

is EXTENDed to 32 bits using the EXT-RI instruction format. This is illustrated in Figure 4-6.

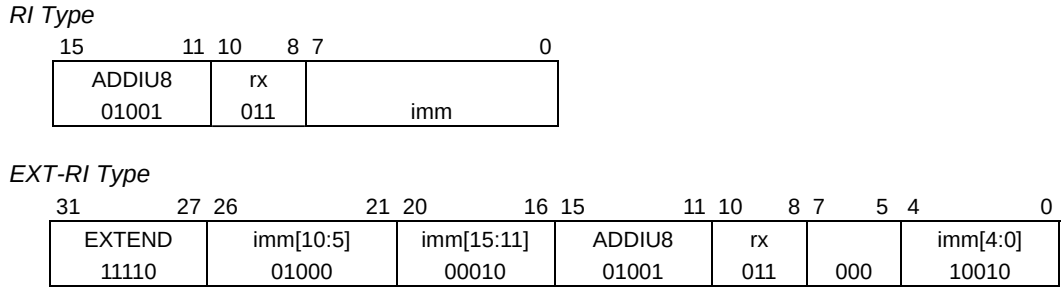


Figure 4-6 RI Format vs. EXT-RI Format

EXTEND extends the immediate fields in the ALU immediate instructions to 16 bits, with one exception. "ADDIU, *ry*, *rx*, *immediate*" has a 4-bit immediate field, but since EXTEND can only supply 11 more bits, the wider immediate is limited to 15 bits.

4.6 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), r29 (sp) and r31 (ra) for specific purposes. These registers are shown as t8, sp and ra. HI and LO are the special registers that hold the results of integer multiply and divide operations.

Instructions Not Implemented in the TX19

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

Table 4-7 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*.
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*.
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	*ry*, *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 *ry*.
	LW	*ry*, *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 *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.
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.
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.

Table 4-8 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 trap 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 trap 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 trap on 2's-complement overflow.
	ADDIU	*rx*, pc, *immediate*	The 8-bit *immediate* is shifted left by two bits and sign-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 trap on 2's-complement overflow.
	ADDIU	*rx*, sp, *immediate*	The 8-bit *immediate* is shifted left by two bits and sign-extended. The resultant value is added to sp and the sum is placed into *rx*. Does not trap 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*.

Table 4-9 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 trap on 2's-complement overflow.
Subtract Unsigned	SUBU	*rz*, *rx*, *ry*	The remainder *rx* - *ry* is placed into *rz*. Does not trap on 2's-complement 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 - *rt* (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-R	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's-complement)
Move	MOVE	*ry*, *r32*	The contents of *r32* is copied into *ry*.
	MOVE	*r32*, *rz*	The contents of *rz* is copied into *r32*.

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

Instruction	Format		Operation
Shift Left Logical	SLL	*rx*, *ry*, *sa*	The contents of *ry* is shifted left by *sa* bits. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into *rx*.
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* is shifted right by *sa* bits. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into *rx*.
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*.
Shift Right Arithmetic	SRA	*rx*, *ry*, *sa*	The contents of *ry* is 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*.
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.

Table 4-11 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.
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.
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.
Move From HI	MFHI *rx*	The contents of register HI is copied to *rx*.
Move From LO	MFLO *rx*	The contents of register LO is copied to *rx*.

Table 4-12 Jump Instructions (16-Bit ISA)

Instruction	Format	Operation
Jump And Link	JAL *target*	A jump is taken 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 + 2. The address of the instruction following the delay slot is saved in r31.
Jump And Link eXchange	JALX *target*	A jump is taken to 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 + 2. 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*	A jump is taken to 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	A jump is taken to 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 And Link Register	JALR ra, *rx*	A jump is taken to 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.

Table 4-13 Branch Instructions (16-Bit ISA)

Instruction	Format	Operation
Branch On Equal To Zero	BEQZ *rx*, *offset*	If *rx* = 0, a branch is taken to the target address specified as a 8-bit *offset* relative to PC + 2.
Branch On Not Equal To Zero	BNEZ *rx*, *offset*	If *rx* ≠ 0, a branch is taken to the target address specified as a 8-bit *offset* relative to PC + 2.
Branch On T8 Equal To Zero	BTEQZ *offset*	If t8 = 0, a branch is taken to the target address specified as a 16-bit *offset* relative to PC + 2.
Branch On T8 Not Equal to Zero	BTNEZ *offset*	If t8 ≠ 0, a branch is taken to the target address specified as a 16-bit *offset* relative to PC + 2.
Branch Unconditional	B *offset*	An unconditionally branch is taken to the target address specified as a 16-bit *offset* relative to PC + 2.

Table 4-14 Special Instructions (16-Bit ISA)

Instruction	Format	Operation
Breakpoint	BREAK *code*	A breakpoint trap occurs, immediately and unconditionally transferring control to the exception handler.
Software Debug Breakpoint Exception	SDBBP *code*	A debug breakpoint trap occurs, immediately and unconditionally transferring control to the exception handler.
Extend	EXTEND *immediate*	The *immediate* is concatenated to the immediate field of the following instruction.

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 TX19 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 require 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.

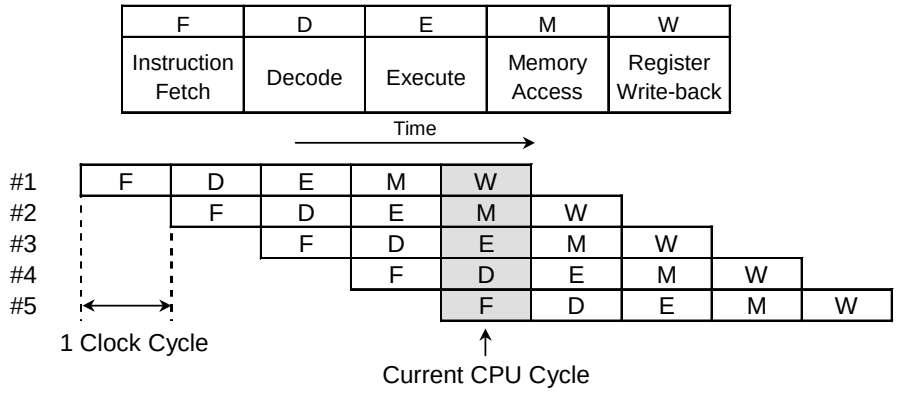


Figure 5-1 Five CPU Pipeline Stages

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 calculates the effective address by adding the offset value to the contents of the base register.
- 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 TX19, 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, nonblocking loads, shared memory synchronization and so on from the view 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	Memory Access	Register Write-back

Figure 5-2 Load Instruction

Load instructions read an operand from memory into a CPU register for subsequent operation by other instructions. In the case of loads from the on-chip fast memory, 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 TX19 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.

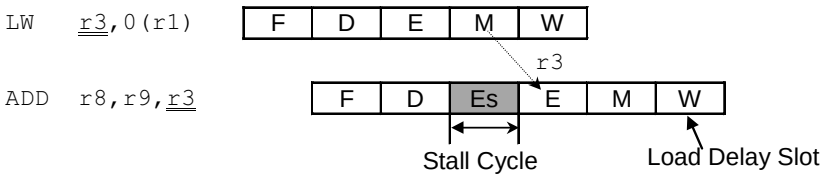


Figure 5-3 Data Dependency Resulting from a Load Instruction

However, this is not a very efficient use of the pipeline. The optimizer, which is executed as 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 part of the code to swap the contents of two memory locations.

- There is data dependency

```
LW r9,0(r8)
LW r10,1(r8)
SW r10,0(r8)  ← Load delay slot
SW r9,1(r8)
```

- There is no data dependency

```
LW r9,0(r8)
LW r10,1(r8)
SW r9,0(r8)  ← Load delay slot
SW r10,1(r8)
```

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

In the rearranged code, the first SW instruction does not depend on the availability of data from the immediately preceding LW instruction. Therefore, the instruction "SW r9, 0(r8)" in the load delay slot for "LW r10, 1(r8)" does not cause a pipeline stall.

5.2.2 Nonblocking 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 TX19 recognizes the presence of data dependency, and if there is no data dependency, it continues to execute subsequent instructions. This is called *nonblocking loads*. By virtue of nonblocking 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 TX19 does not stall on the external memory access resulting from the LW instruction; instead it continues to execute independent instructions (ADD, r6, r4, r2 and ADD r7, r5, r2), and defers execution of a dependent instruction (ADD, r8, r9, r3) until the data has been returned.

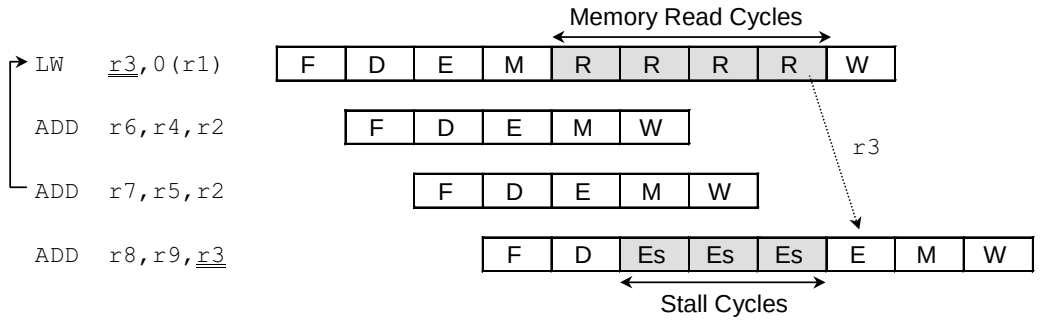


Figure 5-5 Nonblocking Loads

The nonblocking load capability of the TX19 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 by the compiler can yield significant performance improvement.

In nonblocking loads, the Register Write-back (W) stage of the load instruction and a later instruction could attempt to access the on-chip register file simultaneously, causing a resource conflict. In that case, the TX19 inserts a stall cycle into the W stage of the later instruction.

5.2.3 Store Instructions

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

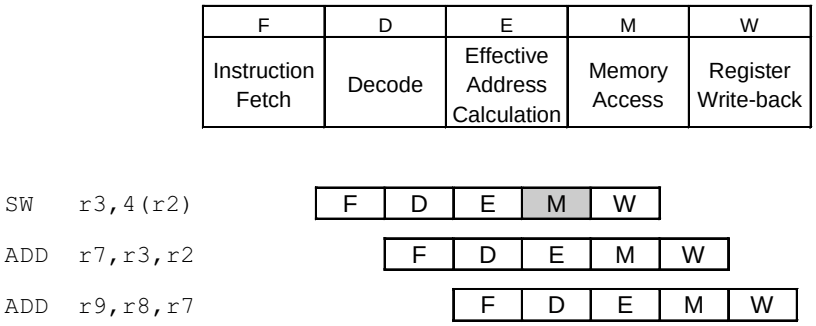


Figure 5-6 Store Instruction

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

5.2.4 SYNC Instruction (32-Bit ISA)

Load and store instructions initiate memory accesses during the M stage. In the meantime, the TX19 continues to execute other instructions in parallel.

The SYNC instruction in the 32-bit ISA provides an ordering function for the effects of load/store and subsequent instructions. Appended to a load or store instruction, 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.

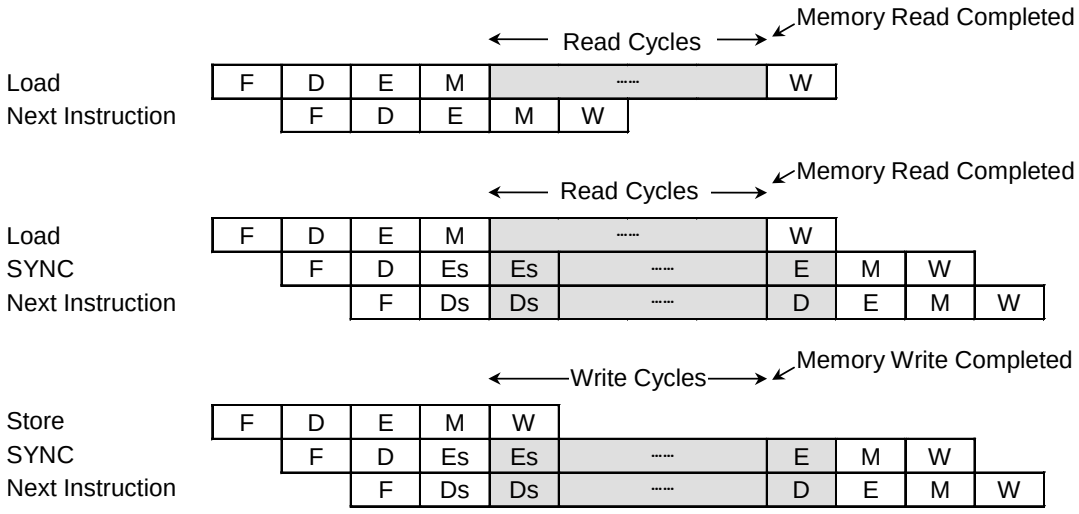


Figure 5-7 SYNC Instruction (32-Bit ISA)

5.3 Jump, Branch and Branch-Likely Instructions

Jump and branch instructions involve a delay or latency of two instructions. 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 5-8 shows how jump and 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 5-8 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.

The jump or branch target address becomes available during the E stage; so it is impossible to perform the fetch of the target instruction without delaying the pipeline. Figure 5-9 show that a jump or branch occurs with a delay of two instructions. In Figure 5-9, the jump or branch delay slot is filled with a useful instruction, thereby reducing the pipeline stall to one cycle. With jump and regular branch instructions, the instruction in the delay slot is always executed prior to the jump/branch taking effect (regardless of whether the branch is taken or not). 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.

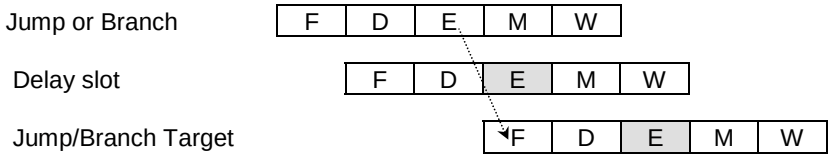


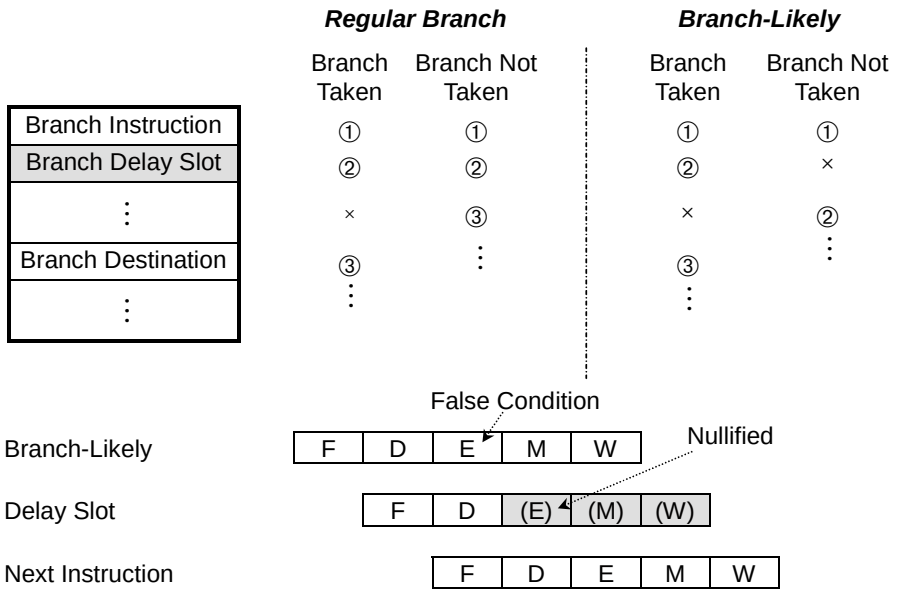
Figure 5-9 Jump and Branch Delay Slots

A jump or branch instruction must not be placed in a jump or branch delay slot. Hardware operation is undefined if that is done.

5.3.2 Branch-Likely Instructions (32-Bit ISA)

A regular branch instruction causes the TX19 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.

On the other hand, a branch-likely instruction causes the TX19 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 5-10).



When a Branch-Likely is Not Taken

Figure 5-10 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 TX19 needs to executes a jump instruction in two steps as shown below. 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.

It is prohibited to place a jump, branch or EXTENDed instruction in the jump delay slot.

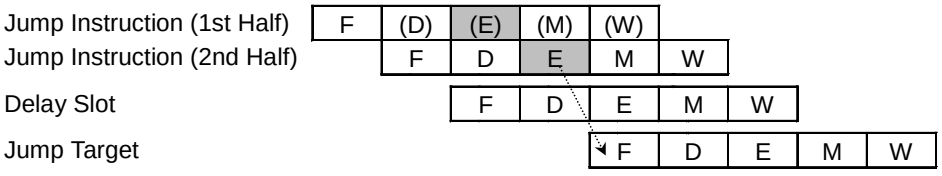


Figure 5-11 Jump Instruction (16-Bit ISA)

5.3.4 Branch Instructions (16-Bit ISA)

Unlike the 32-bit ISA, the 16-bit ISA has no delayed branches (see Figure 5-12). 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.

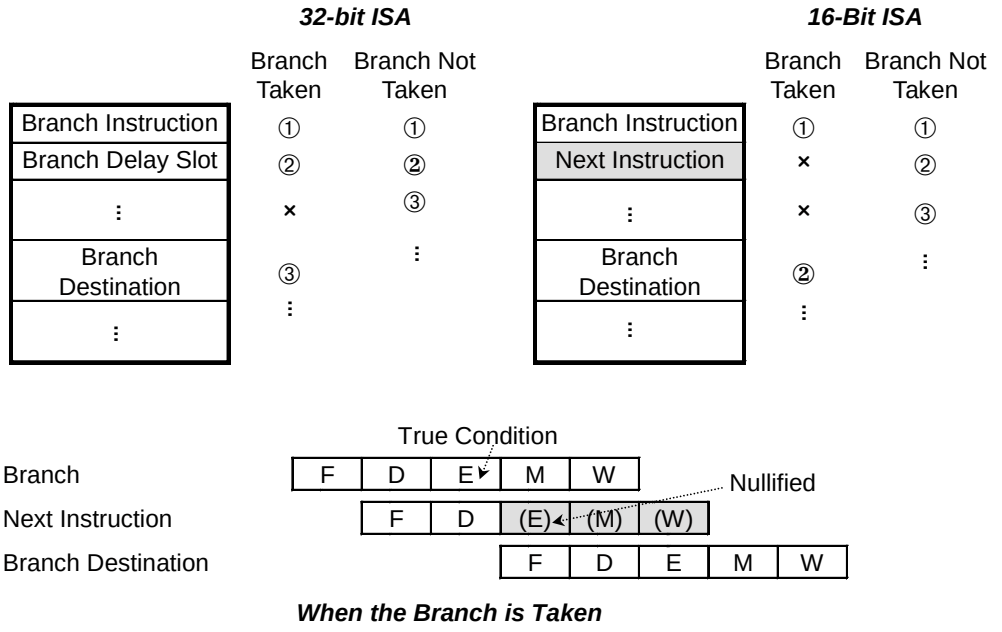


Figure 5-12 Branch Instruction (16-Bit ISA)

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 TX19 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 or MADDU instruction before the quotient and the remainder are available, the pipeline stalls until they do become available.

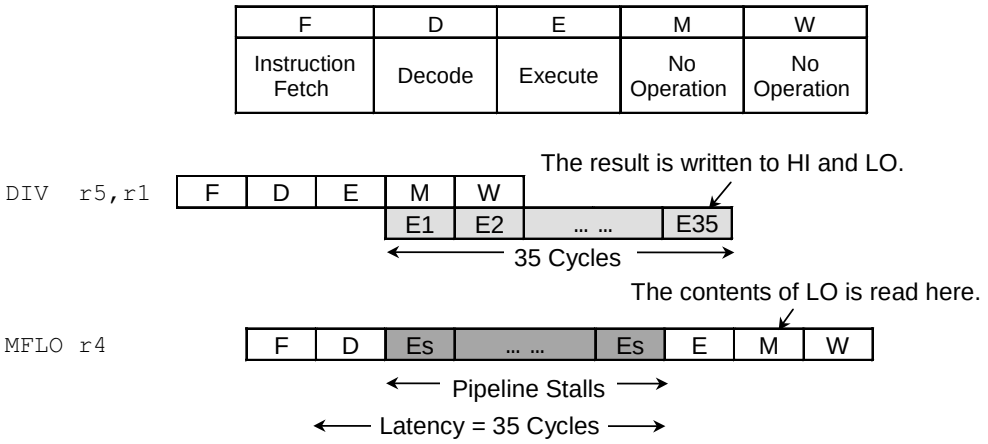


Figure 5-13 Divide Instructions

5.5 Multiply and Multiply-and-Add Instructions

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

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

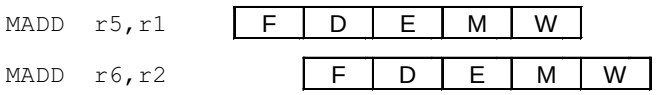


Figure 5-14 Back-to-Back Multiply-and-Add Instructions

The MFHI and MFLO instructions read the contents of the HI and LO registers. Multiply and multiply-and-add instructions can be followed by an MFHI or MFLO instruction without causing pipeline stalls.

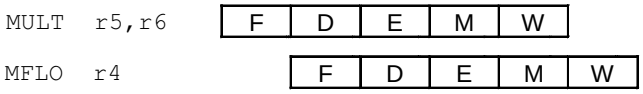


Figure 5-15 Multiply Instruction Followed by an MFLO Instruction

Remember that the result of the multiply and multiply-and-add instructions becomes available after completion of the M stage instead of the E stage. If the multiply or multiply-and-add 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.

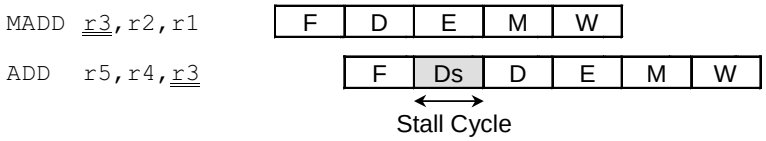


Figure 5-16 Structural Hazard Involving a Multiply Instruction

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. In 16-bit ISA mode, the TX19 executes any EXTENDed instruction in two steps as shown in Figure 5-17.

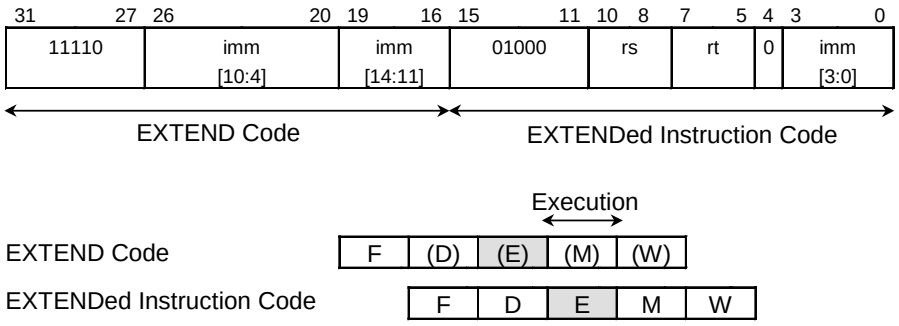


Figure 5-17 EXTENDed Instruction (16-Bit ISA)

Chapter 6 Memory Management

This chapter describes the operating modes of the TX19 processor, the virtual and physical address spaces and how they are mapped.

6.1 Operating Modes

The TX19 has two modes of operation, User mode and Kernel mode. The TX19 enters Kernel mode whenever an exception is taken. Since a Reset exception occurs when a system is reset, the TX19 wakes up in Kernel mode. The processor switches to User mode when the RFE (Restore From Exception) 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. A User-mode program's use of them is restricted. 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 0. The CP0 registers are accessible only when the CU[0] bit in the Status register is 1.

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.

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 is 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 TX19 does not have a cache on-chip, cacheability has no meaning.

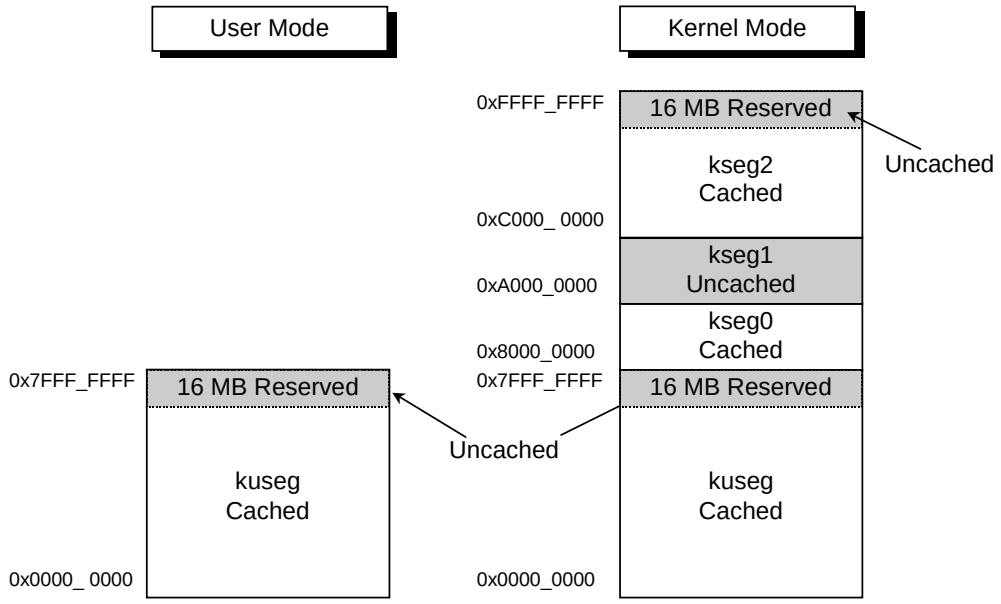


Figure 6-2 Virtual Address Segments

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 Kernel virtual address space 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 6-3.

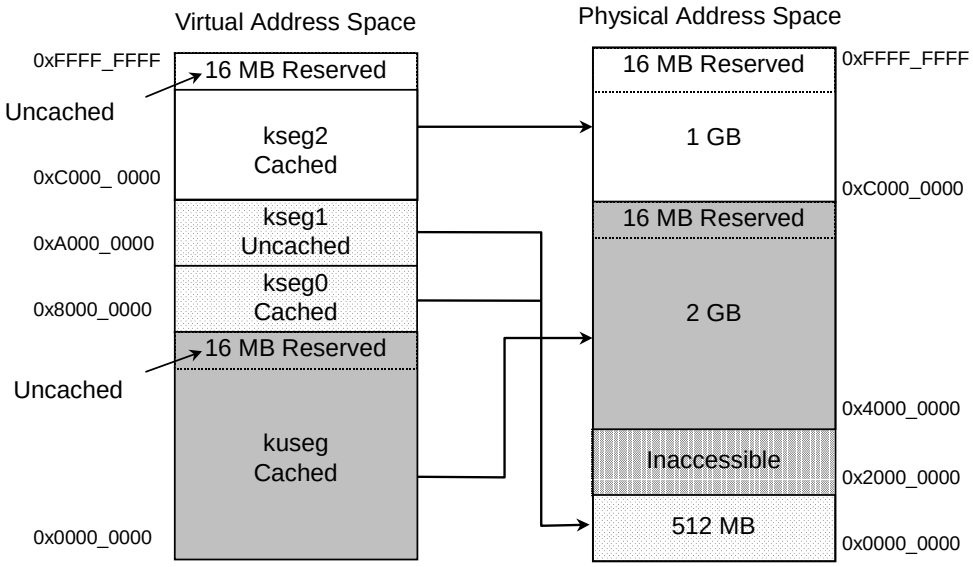


Figure 6-3 Virtual to Physical Address Translation

Figure 6-4 shows the virtual address format used by the TX19. The three highest bits represent segment numbers; only these three bits are involved in virtual-to-physical address translation.

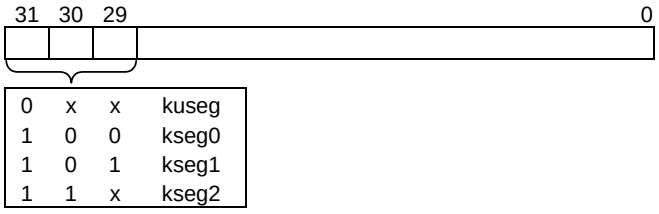


Figure 6-4 Virtual Address Format

- 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 0. 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 6-1 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 the bus interface inside the TX19 core. To maximize performance, the TX19 implements a Harvard architecture, wherein there are two separate sets of address and data buses for code (instructions) and data (operands). Additionally, the TX19 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.

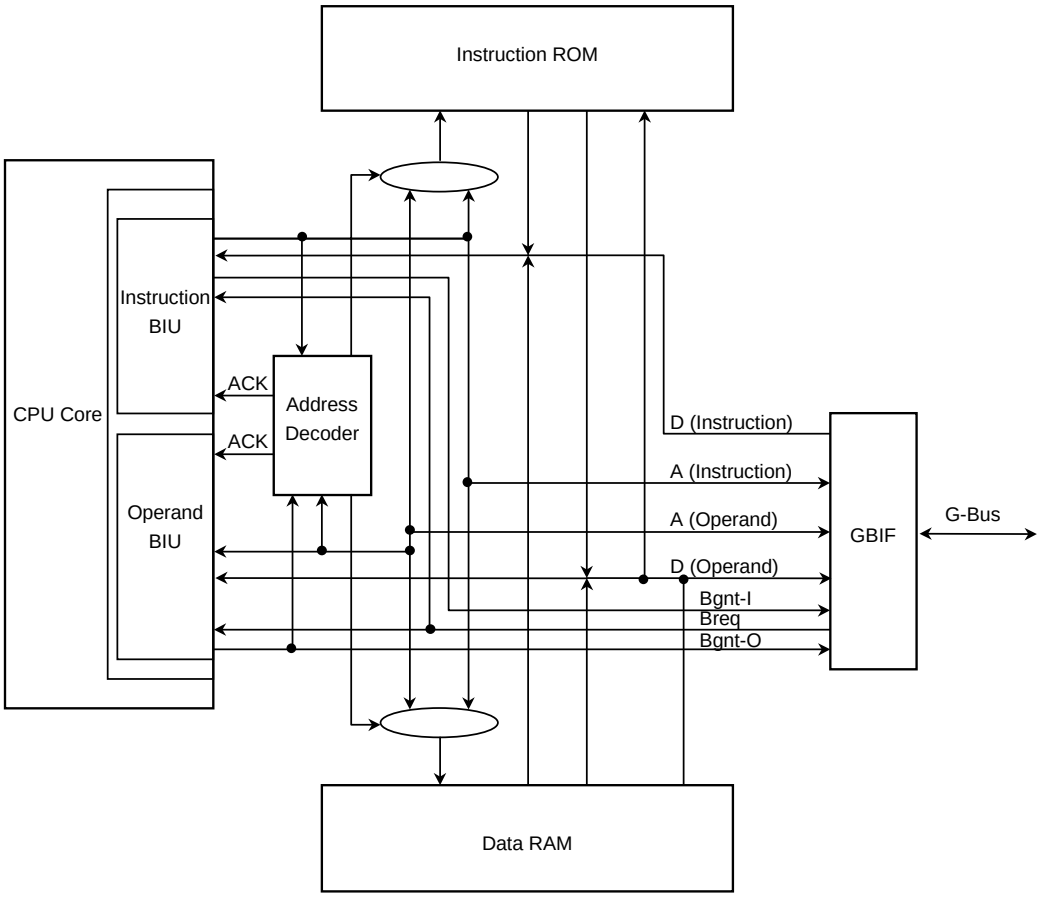


Figure 7-1 General Internal Memory Interface

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 TX19 core features pipelined addressing where it allows up to two outstanding bus cycles at any given time. While the TX19 core waits for the data for the first bus cycle, the address for a second bus cycle is issued. Using pipelined addressing, the TX19 provides support for zero-wait-state reads even for relatively slow memories like flash.

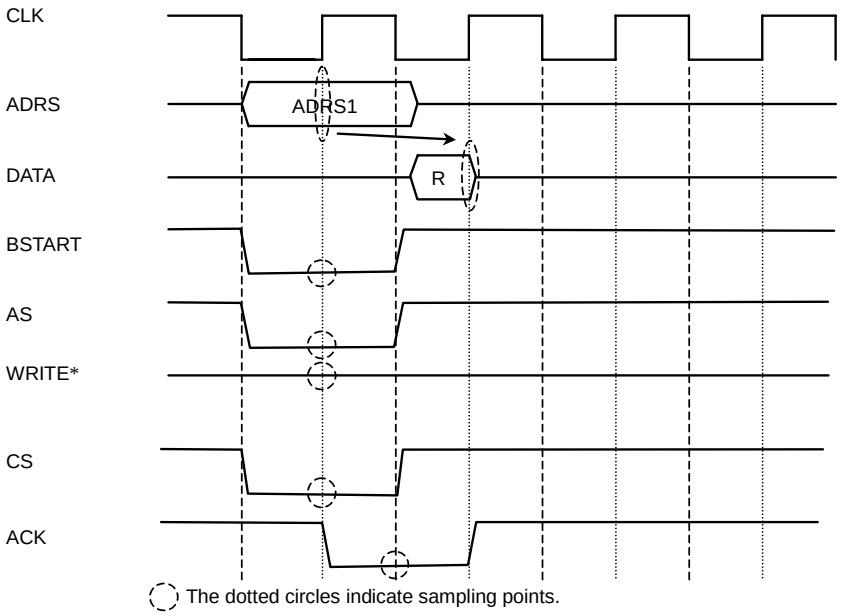


Figure 7-2 Memory Read Timing (Zero-Wait State)

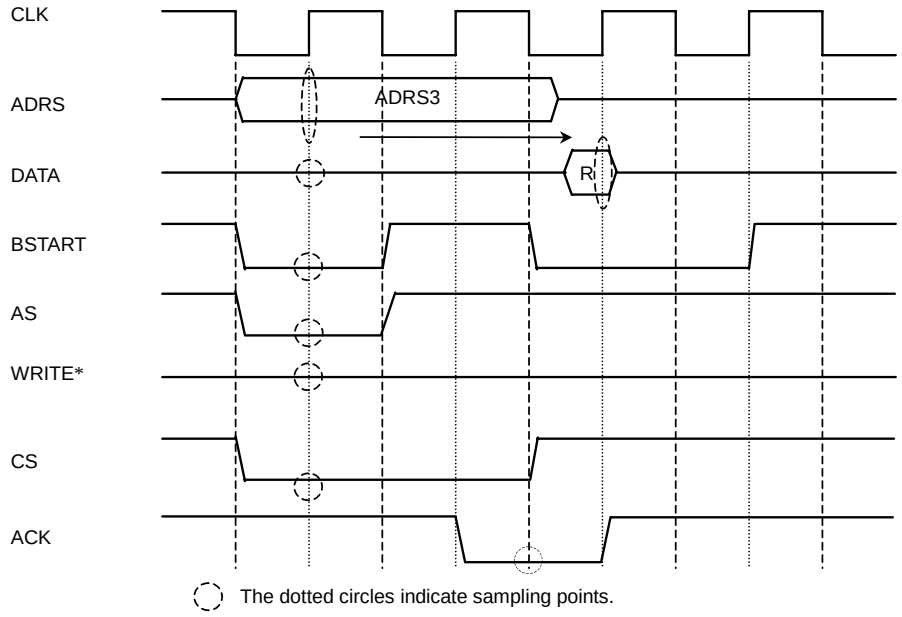


Figure 7-3 Memory Read Timing (1 Wait State for ADRS3)

7.3 Write Operation

Basically, memory write cycles use much the same protocol as memory read cycles. The TX19 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*), etc. are also asserted.

The TX19 core samples the Acknowledge (ACK*) signal on the next falling edge of the system clock after the address is placed on ADRS. If an ACK* is detected, the TX19 goes ahead and issues the address for the next bus cycle. Unless an ACK* is detected, the TX19 inserts a wait state to continue the current bus cycle.

Memory and I/O modules should latch data on the next rising edge of the system clock following the sampling of ACK*.

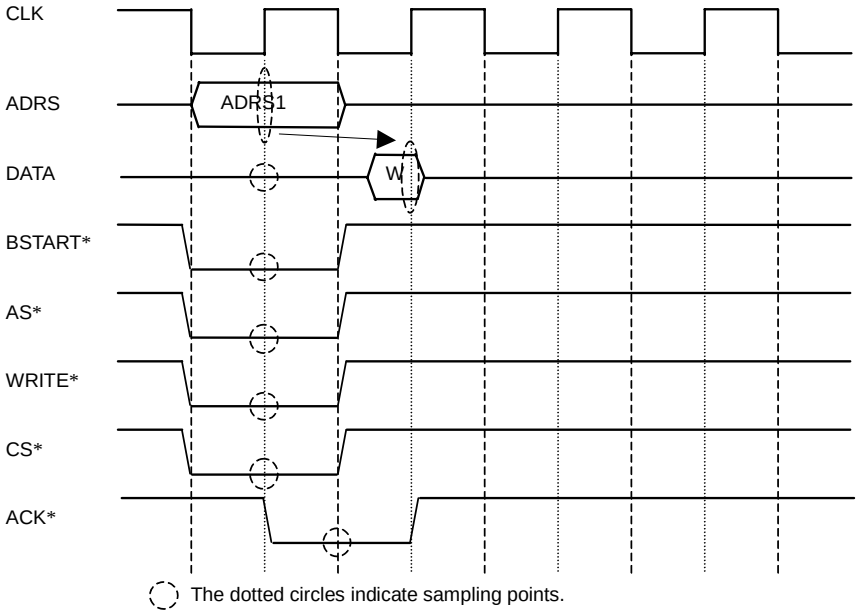


Figure 7-4 Write Timing (Zero-Wait State)

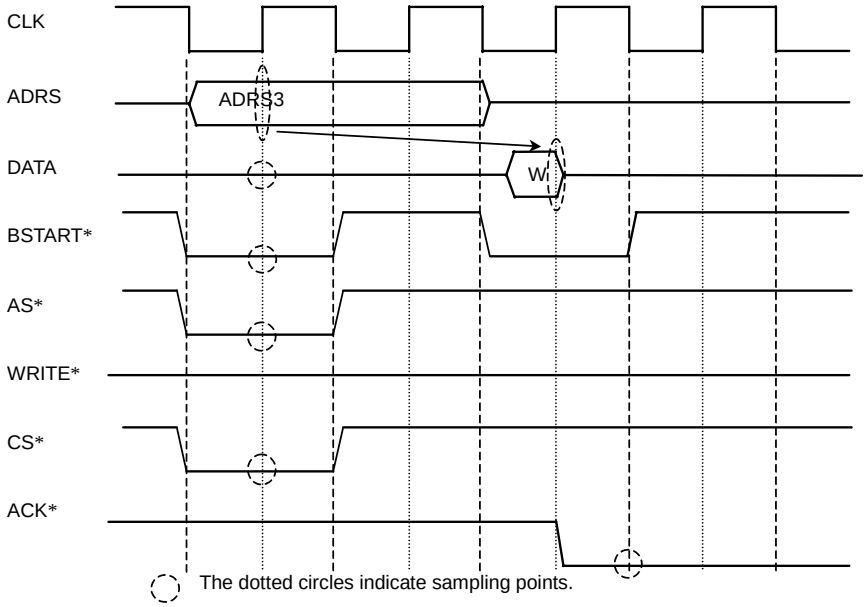


Figure 7-5 Write Timing (1 Wait State for ADR3)

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 CU[0] 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	3	Specifies various configuration options for the TX19 processor.
General Exception Handling	BadVAddr	8	Displays the most recent virtual address that caused a virtual-to-physical address translation error. Read-only.
	Status	12	Contains operating mode (User/Kernel), interrupt enabling and other states of the processor.
	Cause	13	Displays the cause of the last exception.
	EPC	14	Contains the address of the instruction that caused an 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.
	PRId	15	Contains the revision identifier of the TX19 processor. Read-only.
	IE	31	Manipulates the interrupt enable/disable bit in the Status register.
Debug Exception Handling	Debug	16	Displays the cause and the current status of a Debug exception.
	DEPC	17	Contains the address of the instruction that caused a Debug exception, from which point processing resumes after a Debug exception has been serviced.

The sections in this chapter describe the CP0 register organization and how data is represented in these registers. The number following a register name in the headings as in "8.2.1 Config Register (3)" indicates the register number.

8.2 System Configuration Register

The Config register programs various system configuration options for the TX19 processor. It contains the bits for power saving modes (Halt / Doze), reduced frequency modes, data cache enable, instruction cache enable, etc. The TX19 has no on-chip cache; the cache enable bits in the Config register are meaningless. The subsection that follow describes the Config register.

8.2.1 Config Register (3)

Figure 8-1 shows the format of the Config register. Table 8-1 describes the bits in the Config register.

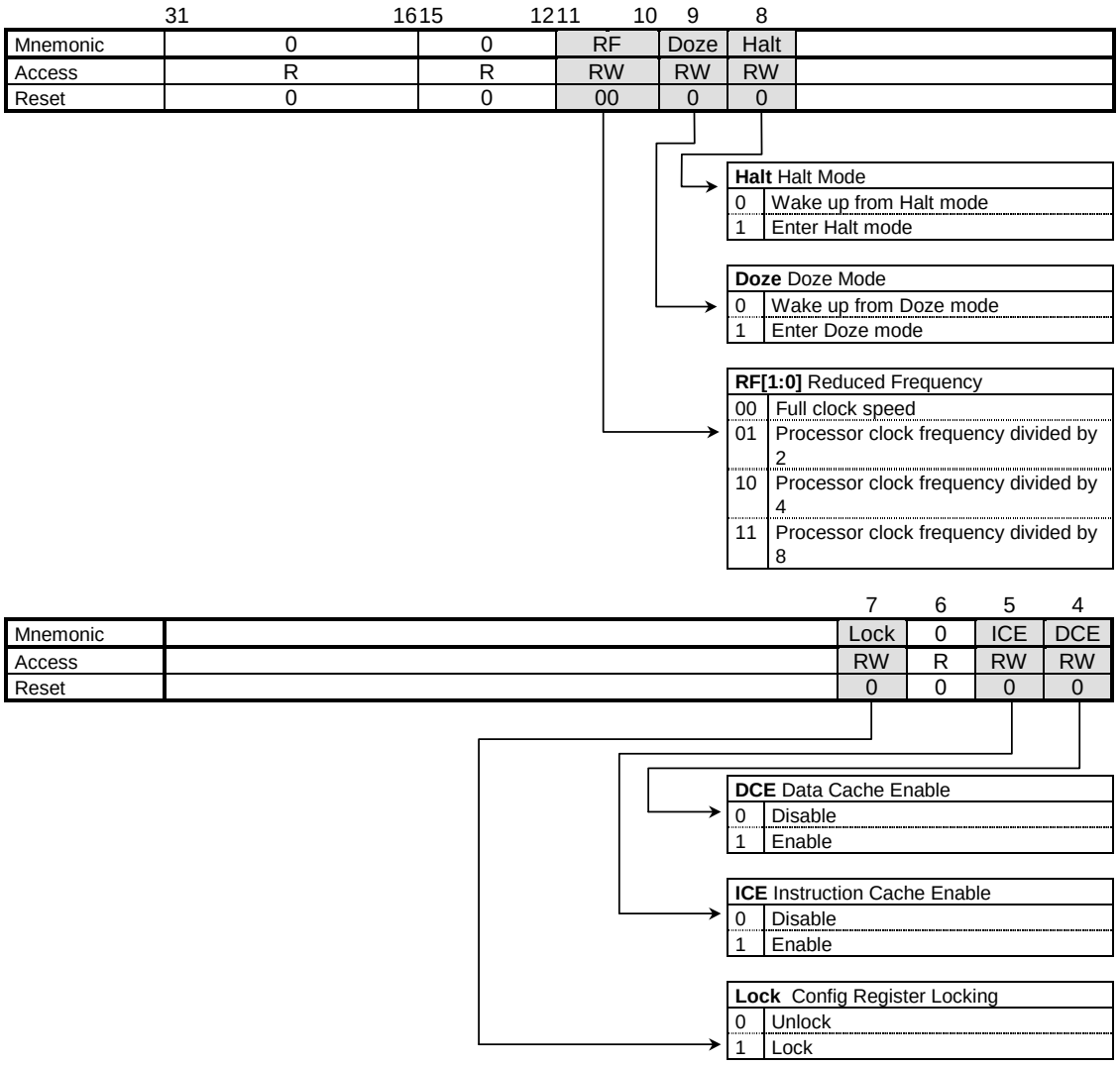


Figure 8-1 Config Register

Table 8-2 Config Register Definition

Mnemonic	Name	Reset Value	Description	Access
RF	Reduced Frequency	00	The value programmed into this field is driven to processor output pins, which are supplied to a clock generator to indicate a clock divisor. See Chapter 10 for details on the reduced frequency modes.	RW
Doze	Doze Mode	0	Enables/disables the Doze mode capability of the TX19. 1 Enter Doze mode. 0 Wake up from Doze mode When set to 1, the CPU freezes the instruction pipeline. Assertion of the reset signal (which initiates a Reset exception), the nonmaskable interrupt signal or the hardware interrupt signal clears this bit, bringing the processor out of Doze mode. (The processor recognizes the interrupt signal even if the interrupt is masked.) See Chapter 10 for details on Doze mode.	RW
Halt	Halt Mode	0	Enables/disables the Halt mode capability of the TX19. 1 Enter Halt mode. 0 Wake up from Halt mode. When set to 1, the CPU freezes the instruction pipeline and ignores any external snoop requests. Assertion of the reset signal (which initiates a Reset exception), the nonmaskable interrupt signal or the hardware interrupt signal clears this bit, bringing the processor out of Halt mode. (The processor recognizes the interrupt signal even if the interrupt is masked.) See Chapter 10 for details on Halt mode.	RW
Lock	Config Register Locking	0	When set to 1, locks the Config register and denies any subsequent write access to it. A Reset exception clears this bit. A Debug exception handler can alter the Config register regardless of the value of the Lock bit if the DM bit in the Debug register is set. Every value carried in an MTC0 instruction is valid, regardless of the value of the Lock bit.	RW
ICE	Instruction Cache Enable	0	Enables/disables the on-chip instruction cache. 1 Enable 0 Disable	RW
DCE	Data Cache Enable	0	Enables/disables the on-chip data cache. 1 Enable 0 Disable	RW
0	Reserved	–	The reserved bits are ignored on write, and read as zero.	R

† The operation is undefined if both the Doze and Halt bits are set simultaneously.

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 Bad Virtual Address (BadVAddr) register is a read-only register that displays the most recent virtual address that caused a virtual-to-physical address translation error. When a translation error occurs, the processor takes an Address Error exception (AdEL or AdES). Figure 8-2 shows the format of the BadVAddr register.

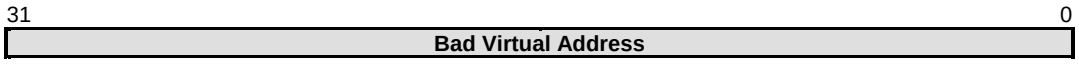


Figure 8-2 BadVAddr Register

8.3.2 Status Register (12)

The Status register contains a three-level stack (current, previous and old) for the Kernel/User mode and interrupt enable bits, and a two-level stack (current and previous) for the interrupt mask level field. The stack is pushed each time an exception is taken and popped by the Restore From Exception (RFE) instruction. The mechanism of these stacks is detailed in Chapter 9, *Exception Handling*. The Status register also contains the bits for coprocessor usability, software interrupt mask and so on. Figure 8-3 shows the format of the Status register. Table 8-3 describes the bits in the Status register.

	31 28	27 26	25 24	23 22	21	20	19	
Mnemonic	CU	0	RE	0	BEV	0	NmI	0
Access	RW	R	RW	R	RW	R	RW	R
Reset	X	0	X	0	1	0	0	0

Note: X signifies undefined.

NmI Nonmaskable Interrupt	
0	Cleared
1	Triggered

BEV Bootstrap Exception Vector	
0	Normal mode: External extended memory mode
1	Alternative mode: On-chip ROM mode

RE Reserved	
0	Must be written as zero

CU[3:0] Coprocessor Usability	
0	Unusable
1	Usable

		18 16	15 13	12	11 8	7 6	5	4	3	2	1	0
Mnemonic		PMask	CMask	0	SwiMask	0	KUo	IEo	KUp	IEp	KUc	IEc
Access		RW	RW	R	RW	R	RW	RW	RW	RW	RW	RW
Reset		X	7	0	X	0	X	X	X	X	0	0

IEc Interrupt Enable, Current	
0	Disable
1	Enable

KUc Kernel/User Mode, Current	
0	Kernel
1	User

IEp Interrupt Enable, Previous	
0	Disable
1	Enable

PMask[2:0] Previous Interrupt Mask Level	
0~7	8-level value

KUp Kernel/User Mode, Previous	
0	Kernel
1	User

CMask[2:0] Current Interrupt Mask Level	
0~7	8-level value

KUo Interrupt Enable, Old	
0	Disable
1	Enable

SwiMask[3:0] Software Interrupt Mask	
0	Disable
1	Enable

KUo Kernel/User Mode, Old	
0	Kernel
1	User

Figure 8-3 Status Register

Table 8-3 Status Register Defintion

Mnemonic	Name	Reset Value	Description	Access
CU[3:0]	Coprocessor Usability	X	Controls the usability of coprocessors units 3 to 0. The CU[3:1] bits control accesses to the respective coprocessors whether 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 CU[0] bit controls the usability of CP0 instructions in User mode. Attempts by a User-mode program to execute a CP0 instruction when the CU[0] bit is cleared causes a Coprocessor Unusable exception. Kernel-mode programs can execute all CP0 instructions, regardless of the setting of the CU[0] bit.	RW
RE	*Reserved*	X	Must be written as zero. When read, zeros are returned.	R
BEV	Bootstrap Exception Vector	1	Set by hardware 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. The BEV bit can be set or cleared by software. When BVE=0, Reset, Nonmaskable Interrupt and Debug exception vectors reside in uncacheable kseg1 space; all the other exception vectors reside in cacheable kseg0 space. See Chapter 9, *Exception Handling*, for details.	RW
Nml	Nonmaskable Interrupt	0	Set when a nonmaskable interrupt signal is asserted low. This bit is cleared by writing a 1.	RW
PMask[2:0] CMask[2:0]	Interrupt Mask Level (Previous / Current)	X7	The Current Interrupt Mask Level field, CMask[2:0], defines the highest priority level that the TX19 ignores. When an interrupt request has a priority higher than the mask level, the TX19 takes an interrupt exception unless the IEc bit is cleared. CMask[2:0] is set to the highest 7 on hardware reset when a Reset exception is initiated. Each time an interrupt exception is taken, the contents of CMask[2:0] is copied into the PMask[2:0] field. When the Restore From Exception (RFE) instruction is executed, the PMask[2:0] value is restored to CMask[2:0]. See Chapter 9, *Exception Handling*, for details.	RW
SwiMask[3:0]	Software Interrupt Mask	X	Used by software to individually enable/disable the four software interrupts. There are four corresponding bits in the Cause register used to generate a software interrupt.	RW
KUo / KUp / KUc	Kernel/User Mode (Old / Previous / Current)	XX0	The KUc bit indicates the current operating mode of the processor, 0=Kernel mode and 1=User mode. The KUo, KUp and KUc bits are a three-level stack (old, previous and current) for the Kernel/User mode. The KUc bit is cleared on hardware reset and when an exception is taken, placing the processor in Kernel mode. Each time an exception is taken, the contents of KUc is pushed to the KUp bit, and the KUp bit is pushed to the KUo bit. When the Restore From Exception (RFT) instruction is executed, the contents of the KUo bit is popped to the KUp bit and the Kup bit is popped to the Kuc bit. The KUo bit remains unchanged. See Chapter 9, *Exception Handling*, for details.	RW

Mnemonic	Name	Reset Value	Description	Access
IEo / IEp / IEc	Interrupt Enable (Old / Previous / Current)	XX0	The IEc bit indicates whether maskable interrupts (hardware and software) are currently enabled or not, 1=enabled and 0=disabled. The IEo, IEp and IEc bits are a three-level stack (old, previous and current) for interrupt enabling. The IEc bit is cleared on hardware reset and when an exception is taken. The IE register can also be used to globally enable or disable interrupts. When an exception is taken, the contents of IEc is pushed to the IEp bit, and the IEp bit is pushed to the IEo bit. When the Restore From Exception (RFT) instruction is executed, the contents of the IEo bit is popped to the IEp bit and the IEp bit is popped to the IEc bit. The IEo bit remains unchanged. See Chapter 9, *Exception Handling*, for details.	RW
0	*Reserved*	0	The reserved bits are ignored on write, and read as zero.	R

8.3.3 Cause Register (13)

The Cause register displays the cause of the last exception. The TX19 recognizes four software interrupts; the Sw[3:0] bits are used by software to set or clear a particular software interrupt. Each of the four software interrupts are vectored to different predetermined locations (see 9.1.3, *Exception Vector Addresses*).

Figure 8-4 shows the format of the Cause register. Table 8-4 describes the bits in the Cause register.

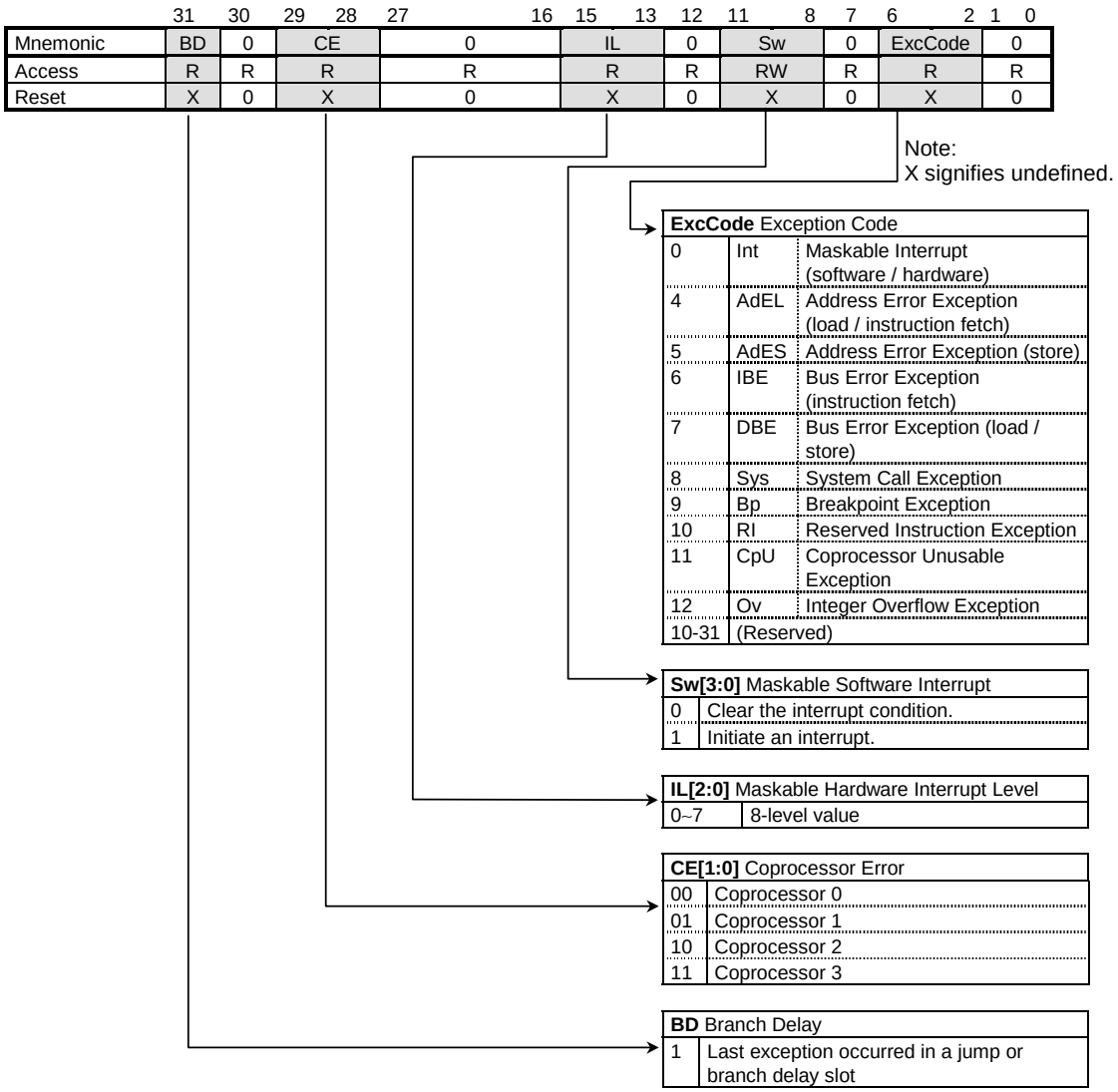


Figure 8-4 Cause Register

Table 8-4 Cause Register Definition

Mnemonic	Name	Reset Value	Description	Access
BD	Branch Delay	X	Set when an exception is taken while the processor is executing an instruction in a jump or branch delay slot.	R
CE[1:0]	Coprocessor Error	X	Indicates the coprocessor unit number referenced when a Coprocessor Unusable exception was taken.	R
IL[2:0]	Interrupt Level	X	Indicates the maskable hardware interrupt priority level. The 3-bit interrupt request signal applied to the processor represents the interrupt priority level and is captured into the IL[2:0] bits irrespective of the interrupt mask level set in the Status register. When an interrupt request has a priority higher than the mask level, the TX19 takes an interrupt exception unless the IEc bit in the Status register is cleared. The IL[2:0] bits are cleared when no interrupt is pending.	R
Sw[3:0]	Maskable Software Interrupt	X	Used by software to set or clear a software interrupt. The TX19 recognizes four software interrupts. There are corresponding interrupt mask bits in the Status register for these interrupts.	R
ExcCode	Exception Code	X	Indicates the cause of the last exception. See Figure 8-4.	RW
0	*Reserved*	–	The reserved bits are ignored on write, and read as zero.	R

8.3.4 EPC Register (14)

The Exception Program Counter (EPC) register saves the contents of the program counter (PC) when an exception is taken. When the instruction is in a jump or branch delay slot, the EPC register is rolled back to point to the jump or branch instruction so that it can be re-executed, and the BD bit in the Cause register is set. As is the case with the PC, the least-significant bit in the EPC register indicates the ISA mode that was in effect when the exception was taken. Figure 8-5 shows the format of the EPC register.

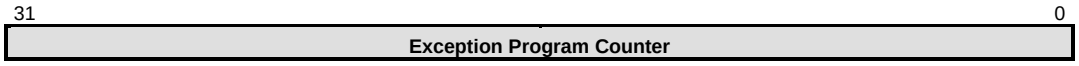


Figure 8-5 EPC Register

8.3.5 PRId Register (15)

The Product Revision Identifier (PRId) register is a read-only register that contains the revision identifier of the processor. Figure 8-6 shows the format of the PRId register. Table 8-5 describes the bits in the PRId register.

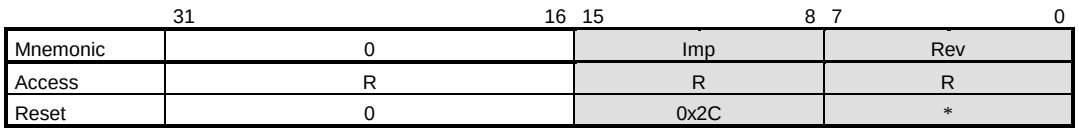


Figure 8-6 PRId Register

Table 8-5 PRId Register Definition

Mnemonic	Name	Reset Value	Description	Access
Imp[7:0]	Implementation Number	0x2C	Contains the execution engine implementation code. The TX19 processor core's implementation code is 0x2C.	R
Rev[7:0]	Revision Number	–	Contains the revision level for this implementation. See hardware user's manuals for the revision number.	R
0	*Reserved*	–	The reserved bits are ignored on write and read as zero.	R

8.3.6 IE Register (31)

The Interrupt Enable (IE) register is used to set or clear the IEc bit in the Status register to enable or disable interrupts. Writing a zero to the IE register causes the IEc bit in the Status register to be cleared; writing a non-zero value to the IE register causes the IEc bit to be set. Use the instruction "MTC0 r0, IE" to disable interrupts. Use a register that contains a non-zero value as the target register (*rt*) like "MTC0 $sp, IE" to enable interrupts. Figure 8-7 shows the format of the IE register.

Figure 8-7 IE Register

You can also set or clear the IEc (Interrupt Enable) bit of the Status register directly. However, to do this, you need to use a sequence of several instructions as shown below:

```
MFC0 r26,C0_STATUS
NOP
OR r26,r26,SR_IEC
MTC r26,C0_STATUS
```

where, C0_STATUS represents the Status register and SR_IEC a constant (0x0000_0001). (These are typically defined in a header file for the assembler.) In contrast to executing the above code, the IE register provides for fast enabling/disabling of interrupts.

8.4 Debug Exception Handling Registers

The TX19 allows program instruction execution to arbitrarily stop to handle debugging events. The TX19 incorporates extra hardware-based features to enhance program debug.

8.4.1 Debug Register (16)

As a debugging aid, the Debug register reflects conditions that were in effect at the time the Debug exception occurred. It also allows you to initiate debug processing. Code execution breakpoints can be generated by embedding Software Debug Breakpoint (SDBBP) instructions in the code. Additionally, the single-step feature may be enabled by setting the SSt bit in the Debug register. Figure 8-8 shows the format of the Debug register. Table 8-6 describes the bit in the Debug register.

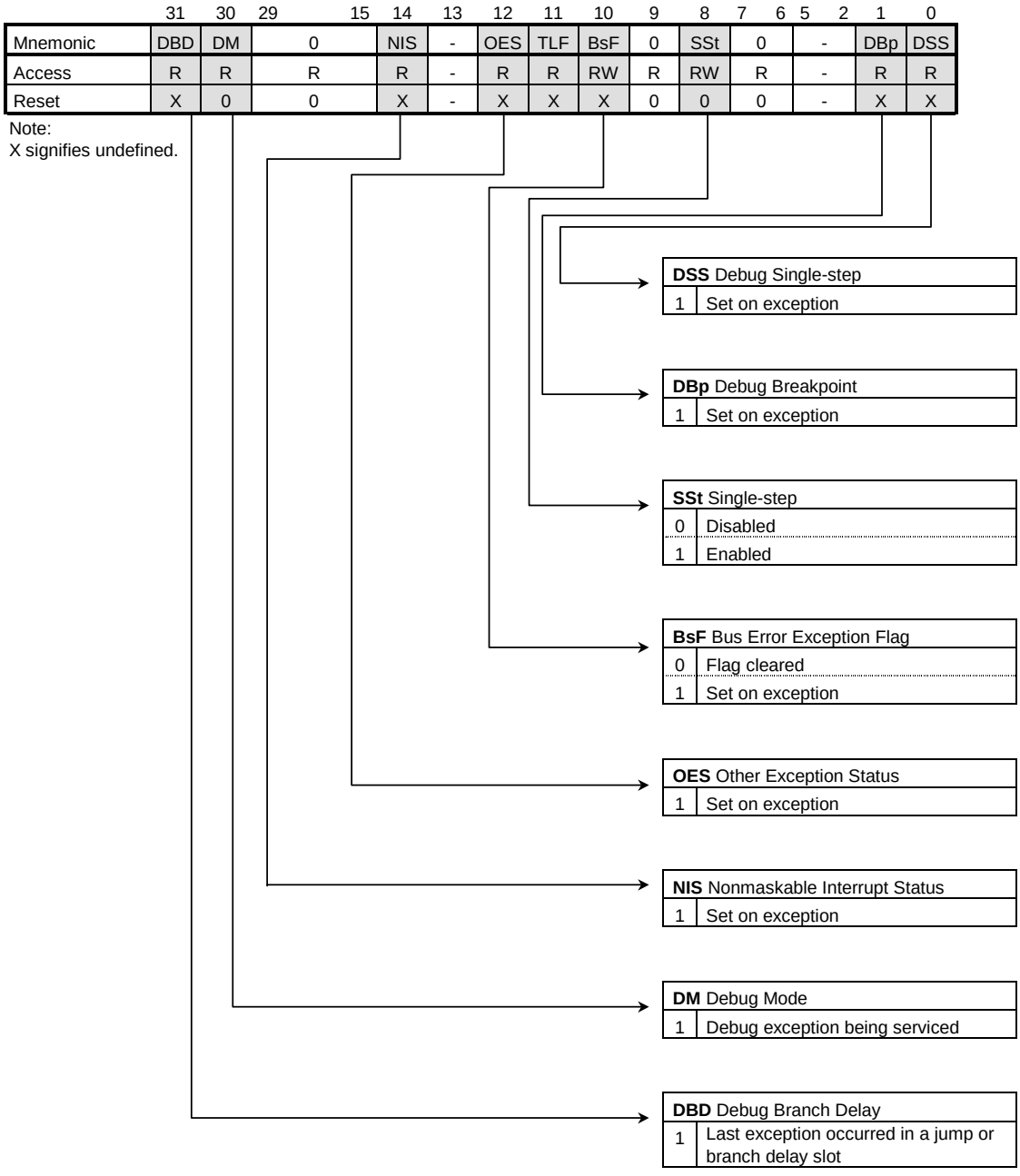


Figure 8-8 Debug Register

Table 8-6 Debug Register Definition

Mnemonic	Name	Reset Value	Description	Access
DBD	Debug Branch Delay	X	Set when a Debug exception is taken while the processor is executing an instruction in a jump or branch delay slot.	R
DM	Debug Mode	0	Set while the Debug exception is being serviced. Cleared by the Debug Exception Return (DERET) instruction.	R
NIS	Nonmaskable Interrupt Status	X	Set if a Debug exception and a Nonmaskable Interrupt exception occurred simultaneously. At this point, the Status, Cause, EPC and BadVAddr registers reflect conditions after the Nonmaskable Interrupt exception was taken, but the DEPC register is not loaded with the vector address of the Nonmaskable Interrupt exception (0xBFC0_0000) yet.	R
OES	Other Exception Status	X	Set if a Debug Exception and a general exception other than the Reset and Nonmaskable Interrupt exceptions occurred simultaneously. At this point, the Status, Cause, EPC and BadVAddr registers reflect conditions after the general exception was taken, but the DEPC register is not loaded with the general exception vector address yet.	R
BsF	Bus Error Exception Flag	X	Set if a Bus Error exception occurred while the Debug exception was being serviced. Writing a zero clears this bit.	RW
SSt	Single-step	0	Enables/disables single-step execution. Once set, a Single-step Exception occurs after the next instruction completes execution. The DM bit, when set, overrides this bit setting.	RW
DBp	Debug Breakpoint	X	Set if a Debug Breakpoint exception occurred.	R
DDS	Debug Single-step	X	Set if a Single-step exception occurred.	R
0	*Reserved*	0	The reserved bits are ignored on write, and read as zero.	R
–	*Reserved*	X	Reserved for future use.	R
TLF	*Reserved*	X	Reserved for future use.	R

8.4.2 DEPC Register (17)

The Debug Exception Program Counter (DEPC) saves the contents of the program counter (PC) when a Debug exception is taken. When the instruction is in a jump or branch delay slot, the DEPC 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 in the DEPC register indicates the ISA mode that was in effect when the exception was taken. Figure 8-9 shows the format of the DEPC register.

Figure 8-9 DEPC Register

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 TX19 are referred to as either general exceptions or debug exceptions. This section explains all types of exceptions other than debug exceptions which are used exclusively for program debug. It provides 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.

For all exceptions other than a Reset exception, exception processing occurs in the sequence shown in Figure 9-1.

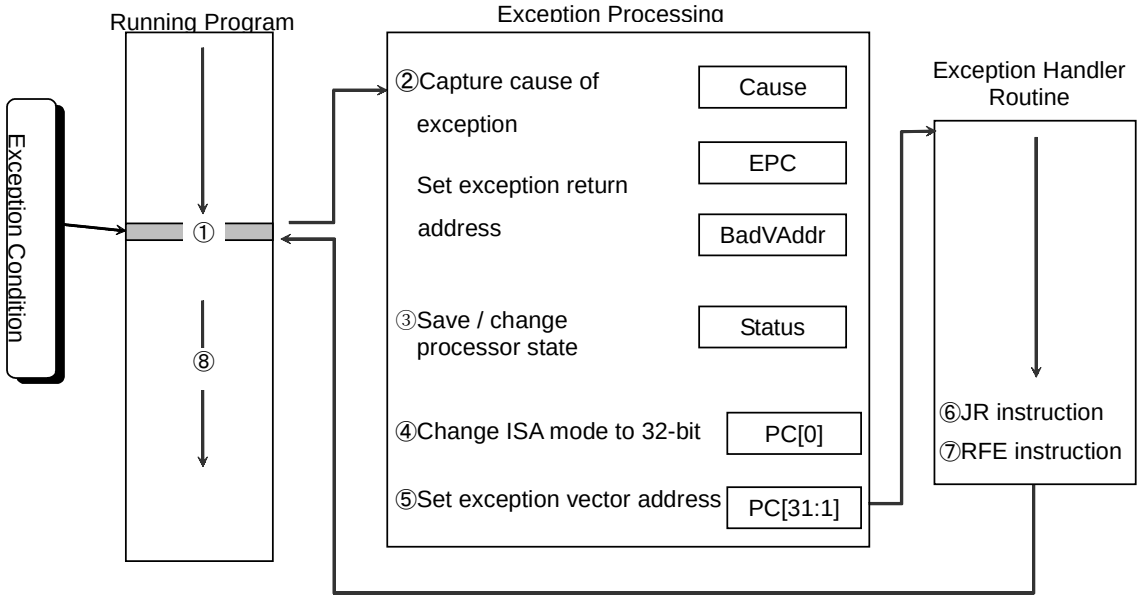


Figure 9-1 Exception Operation

The following paragraph numbers are keyed to the call-out numbers in Figure 9-1.

1. The currently executing instruction and any subsequent instructions in the pipeline are aborted.
2. The Cause register captures information about the cause of the exception. Although multiple exception conditions map to a single exception vector, a more specific condition can be determined by examining the Cause register.

   The EPC register captures the virtual address of the instruction that caused an exception, from which point processing resumes after the exception has been serviced. When the instruction is in a jump or branch delay slot, the EPC is rolled back to point to the jump or branch instruction so that it can be re-executed, and the BD bit in the Cause register is set. The least-significant bit of the EPC register is the ISA mode bit that indicates the ISA mode that was in effect when the exception occurred.

   If the exception is an Address Error, the BadVAddr register captures the virtual address that caused a virtual-to-physical address translation error.
3. The Status register captures information about the current operating state of the processor and then causes the processor to enter Kernel mode for exception processing and turn off all interrupts.
4. If the exception occurred in 16-bit ISA mode, the least-significant bit (i.e., the ISA mode bit) of the Program Counter (PC) is set to zero, bringing the processor into 32-bit ISA mode.
5. The PC is loaded with the exception vector address to jump to the starting location of the exception handler.
6. After the exception has been serviced, the Jump Register (JR) instruction is used to jump back to the return address.
7. At the end of the exception handler routine is the Restore From Exception (RFE) instruction to

restore the system context to the state that existed before the exception. The RFE instruction is actually executed in the jump delay slot prior to the JR instruction.

8. Processing resumes from the point where the processor left off when the exception occurred.

9.1.2 General Exception Types

Figure 9-1 gives the types of general exceptions that can occur in the TX19 processor. The ExcCode field in the Cause register indicates the cause of the most recent exception. Later subsections describe each of these exceptions in greater details in this order.

Table 9-1 General Exception Types

ExcCode	Exception Type	Mnemonic	Description
0	Maskable Interrupt	Int	A Maskable Interrupt exception occurs when the interrupt signal with a priority level higher than the value of the CMask[2:0] field in the Status register is delivered or one of the Sw[3:0] bits in the Cause register is set by software.
-	Nonmaskable Interrupt	Nml	A Nonmaskable Interrupt exception occurs when the NMI* signal is asserted low.
4	Address Error (Load)	AdEL	An Address Error exception is caused by the following events: • Loads from unaligned addresses • Stores to unaligned addresses • 32-bit instruction fetches from addresses not on word boundaries • User-mode accesses to the privileged Kernel address spaces
5	Address Error (Store)	AdES	
6	Bus Error (Instruction Fetch)	IBE	A Bus Error exception occurs when the bus error signal is asserted during bus cycles.
7	Bus Error (Data)	DBE	
8	System Call	Sys	A System Call exception occurs when a SYSCALL instruction is executed.
9	Breakpoint	Bp	A Breakpoint exception occurs when a BREAK instruction is executed.
10	Reserved Instruction	RI	A Reserved Instruction exception occurs when execution of an instruction is attempted with an undefined or reserved major or minor opcode.
11	Coprocessor Unusable	CpU	A Coprocessor Unusable exception occurs when an attempt is made to execute a coprocessor instruction in User mode when the corresponding CU bit in the Status register is cleared.
12	Integer Overflow	Ov	An Integer Overflow exception is caused by an add or subtract instruction on 2's-complement overflow.
-	Reset	Reset	A Reset exception occurs when the reset signal is asserted and then deasserted.
-	Debug	-	See 9.3, *Debug Exceptions*.

† The mnemonic for the Debug exception is not defined.

9.1.3 Exception Vector Addresses

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_200. Values of the other vectors depend on the BEV (Bootstrap Exception Vector) bit of the Status register. Table 9-2 shows the exception vector addresses.

Table 9-2 Exception Vector Addresses

Exception Type		Vector Address			
		BEV=0		BEV=1	
		Virtual	Physical	Virtual	Physical
Reset Nonmaskable Interrupt		0xBFC0_0000	0x1FC0_0000	0xBFC0_0000	0x1FC0_0000
Debug		0xBFC0_0200	0x1FC0_0200	0xBFC0_0200	0x1FC0_0200
Maskable Interrupts	Software Interrupt Swi0	0x8000_0110	0x0000_0110	0xBFC0_0210	0x1FC0_0210
	Software Interrupt Swi1	0x8000_0120	0x0000_0120	0xBFC0_0220	0x1FC0_0220
	Software Interrupt Swi2	0x8000_0130	0x0000_0130	0xBFC0_0230	0x1FC0_0230
	Software Interrupt Swi3	0x8000_0140	0x0000_0140	0xBFC0_0240	0x1FC0_0240
	Hardware Interrupt	0x8000_0160	0x0000_0160	0xBFC0_0260	0x1FC0_0260
General Exceptions		0x8000_0080	0x0000_0080	0xBFC0_0180	0x1FC0_0180

The BEV bit in the Status register is set by hardware 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. The BEV bit can be set or cleared by software. When BVE=0, Reset, Nonmaskable Interrupt and Debug exception vectors reside in uncacheable kseg1 space, but all the other exception vectors reside in cacheable kseg0 space.

9.1.4 General Exception Priorities

While more than one exception can occur at a time, the TX19 reports only one exception with the priority order shown in Table 9-3.

Table 9-3 Exception Priorities

Priority	Exception Type	Mnemonic
Highest	Reset	Reset
	Bus Error (Instruction Fetch)	IBE
	Bus Error (Data Access)	DBE
	Nonmaskable Interrupt	Nml
	Address Error (Instruction Fetch)	AdEL
	Coprocessor Unusable	CpU
	Integer Overflow, System Call, Breakpoint, Reserved Instruction	Ov, Sys, Bp, RI
	Address Error (Load)	AdEL
	Address Error (Store)	AdES
Lowest	Maskable Interrupt	Int

9.1.5 Saving and Restoring Processor Context

The Status register contains a three-level stack (current, previous and old) for the Kernel/User Mode and Interrupt Enable bits. The KUc bit indicates the current operating mode of the processor, 0=Kernel mode and 1=User mode. The IEc bit indicates whether maskable interrupts, both hardware and software, are currently enabled or not, 1=enabled and 0=disabled.

When an exception occurs, the KUc and IEc bits are pushed to the "previous" bits (KUp/IEp) and the Kup and IEp bits are pushed to the "old" bits (KUo/KEo). The "current" bits (KUc/IEc) are cleared so the processor enters Kernel mode and disables all interrupts.

This three-level stack within the Status register allows the processor to respond to two levels of exceptions before software must save the contents of the Status register to a general-purpose register or stack in memory.

After an exception has been serviced, processor context must be restored to the state that existed prior to the exception. The RFE instruction is used to do this. When the RFE instruction is executed, the contents of the "old" bits (KUo/IEo) are popped to the "previous" bits (KUp/IEp) and the "previous" bits (KUp/IEp) are popped to the "current" bits (KUc/IEc). The "old" bits (KUo/IEo) remain unchanged.

Additionally, the Status register provides a two-level stack for the Interrupt Mask Level field (previous and current). The three-bit CMask[2:0] field defines the highest priority level that the processor ignores. When an interrupt request has a priority higher than the mask level, the processor takes an interrupt exception. When an exception is taken, the contents of the CMask[2:0] field is pushed to the "previous" field, PMask[2:0]. The RFT instruction restores the PMask[2:0] value to CMask[2:0].

Figure 9-2 shows how the processor manipulates the Status register during exception recognition and how the Status register bits are restored by the RFT instruction after exception processing.

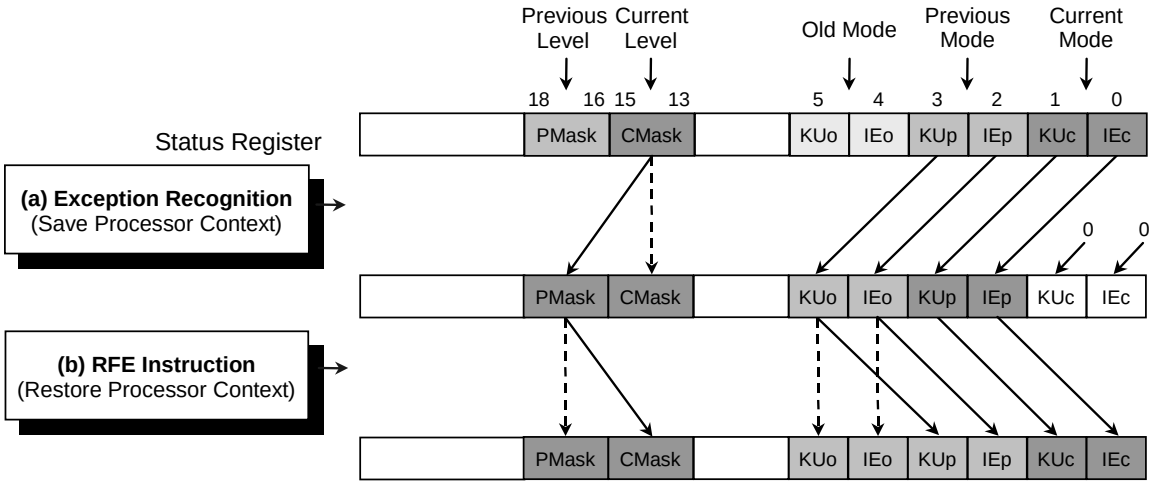


Figure 9-2 Kernel and Interrupt Enable Bits

When an exception occurs, the EPC register captures the virtual address of the instruction that caused an exception. When the instruction was in a jump or branch delay slot, the EPC register is rolled back to point to the jump or branch instruction so that it can be re-executed. The least-significant bit in the EPC register saves the ISA mode that was in effect prior to the exception.

Typically, exception handlers saves the Cause, Status, BadVAddr and EPC registers in general registers or onto stack in memory to preserve processor context. This does have the advantage that interrupts can be re-enabled while the original exception is being handled, thus allowing a priority interrupt model to be built. When the processor takes an exception, subsequent interrupts are automatically disabled; so it is possible to execute an exception handler, leaving the processor context in the CP0 registers. However, in this case, care must be taken to ensure that the execution of the exception handler does not generate any other exception.

After the exception has been serviced, the JR instruction is used to jump to the address at which the exception occurred. Since the JR instruction takes only a general-purpose register as its operand, the return address must be set into a general-purpose register before execution of a JR instruction. The JR instruction restores both the return address and ISA mode bit into the PC.

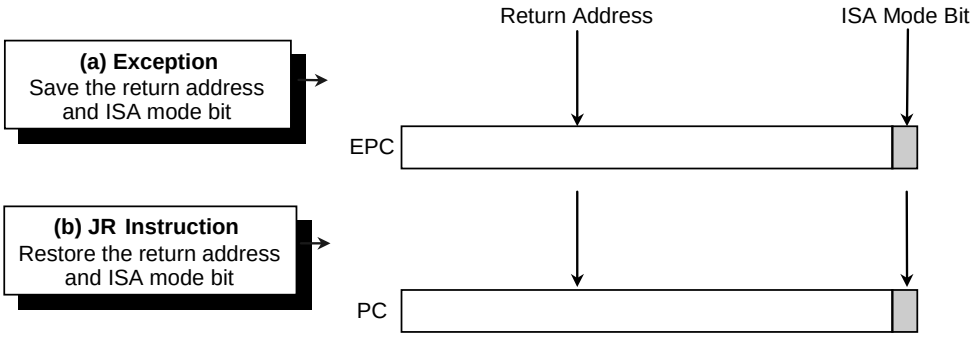


Figure 9-3 Saving and Restoring ISA Mode

9.1.6 Maskable Interrupt Exception

Cause

This exception occurs when one of the maskable interrupt conditions (software or hardware) occurs. Section 9.2 *Interrupts*, describes how the processor recognizes interrupts.

Handling

Figure 9-4 highlights the CP0 register fields that are used to handle this exception.

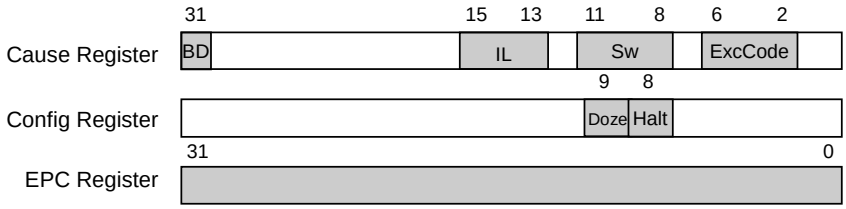


Figure 9-4 Maskable Interrupt Exception

1. The Int code (0) is set into the ExcCode field in the Cause register.
2. If external hardware generated the interrupt, the IL field in the Cause register shows its priority level. If software generated the interrupt, the Sw field shows which of the software interrupts are pending; more than one interrupts may be pending at a time.
3. If the interrupt is hardware-generated, the Halt and Doze bits in the Config register are cleared.
4. The EPC register stores the program counter (PC) on the interrupt. If the interrupt-causing instruction 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 in the EPC register saves the ISA mode that was in effect prior to the exception.
5. Processor context in the Status register is stacked, and the KUc and IEc bits are cleared to enter Kernel mode and disable all interrupts (see 9.1.5, *Saving and Restoring Processor Context*).
6. If the exception occurs while the processor is in 16-bit ISA mode, the processor switches to 32-bit ISA mode.
7. If the interrupt is hardware-generated, the processor jumps to the exception handler located at address 0x8000_0160. If the interrupt is generated by software, the processor jumps to the corresponding exception vector address.
8. If the interrupt is hardware-generated, the exception handler should access the interrupt vector register in the peripheral interrupt controller to determine the source of the interrupt and transfer control to an appropriate interrupt service routine. At this time, the interrupt request level is set to the CMask field in the Status register. If the interrupt request changes to a lower level before the interrupt vector register is read, the interrupt might not be processed properly.

If software generates the interrupt, clear the interrupt condition by setting the corresponding Sw bit in the Cause register. If external hardware generates the interrupt, clear the interrupt condition by removing the conditions that caused the processor's interrupt pin to be asserted.

9.1.7 Nonmaskable Interrupt Exception

Cause

This exception occurs when the processor's nonmaskable interrupt pin is asserted.

Handling

Figure 9-5 highlights the CP0 register fields that are used to handle this exception.

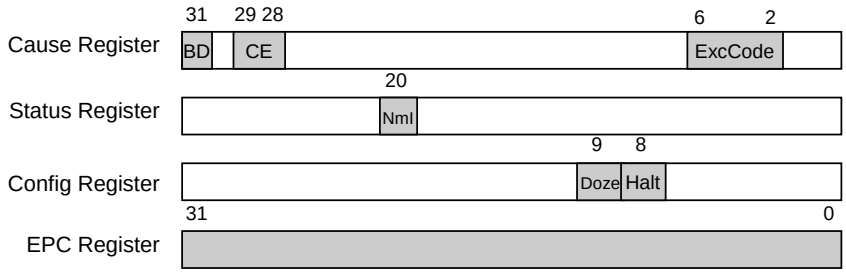


Figure 9-5 Nonmaskable Interrupt Exception

1. The Exception Code (ExeCode) and Coprocessor Error (CE) bits in the Cause register are set to X.
2. The Nonmaskable Interrupt (Nml) bit in the Cause register is set.
3. The Halt and Doze bits in the Config register are cleared.
4. The EPC register stores the program counter (PC) on the interrupt. If the processor is executing an instruction 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 in the EPC register saves the ISA mode that was in effect prior to the exception.
5. Processor context in the Status register is stacked, and the KUc and IEc bits are cleared to enter Kernel mode and disable all interrupts (see 9.1.5, *Saving and Restoring Processor Context*).
6. If the exception occurs while the processor is in 16-bit ISA mode, the processor switches to 32-bit ISA mode.
7. The processor jumps to the exception handler located at address 0xBFC0_0000.

When a nonmaskable 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.

9.1.8 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
- fetch a 16-bit ISA instruction that is not aligned on a halfword boundary
- load or store a word that is not aligned on a word boundary
- load or store a halfword that is not aligned on a halfword boundary
- reference a Kernel-mode address space (kseg0, kseg1 or kseg2) in User mode

During instruction fetches, any instruction can generate an Address Error exception. The LB, LBU, LH, LHU, LW, LWL, LWR, SB, SH, SW, SWL and SWR instructions can generate an Address Error exception due to one of the other causes.

Handling

Figure 9-6 highlights the CP0 register fields that are used to handle this exception.

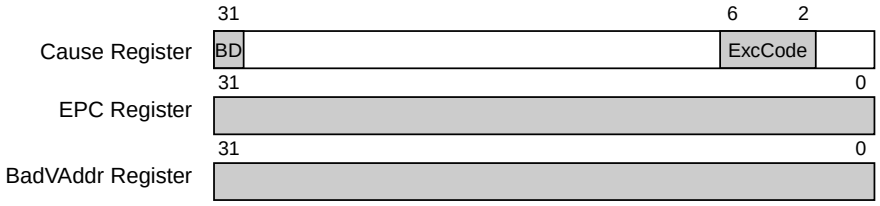


Figure 9-6 Address Error Exception

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 EPC register stores the program counter on the exception. If the processor is executing an instruction 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 in the EPC register saves the ISA mode that was in effect prior to the exception.
3. The BadVAddr register stores the virtual address that is not properly aligned or the virtual address that improperly addressed a Kernel-segment address while in User mode.
4. Processor context in the Status register is stacked, and the KUc and IEc bits are cleared to enter Kernel mode and disable all interrupts (see 9.1.5, *Saving and Restoring Processor Context*).
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 0x8000_0080.

9.1.9 Bus Error Exception

Cause

This exception occurs when an assertion of the bus error signal is acknowledged during memory bus cycles.

During instruction fetches, any instruction can generate a Bus Error exception. The LB, LBU, LH, LHU, LW, LWL, LWR, SB, SH, SW, SWL and SWR instructions can generate a Bus Error exception during a load or store operation.

Handling

Figure 9-7 highlights the CP0 register fields that are used to handle this exception.

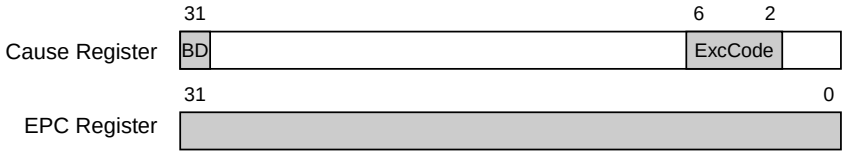


Figure 9-7 Bus Error Exception

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 EPC register saves the program counter 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.

   For all the other cases such as bus time-outs and backplane bus parity errors, the EPC register is set to X. If there is a need to know the address of the exception-causing instruction, external hardware must provide a mechanism to save it.
3. Processor context in the Status register is stacked, and the KUc and IEc bits are cleared to enter Kernel mode and disable all interrupts (see 9.1.5, *Saving and Restoring Processor Context*).
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 0x8000_0080.

Bus error signaling causes the ongoing memory bus cycle to be aborted immediately. In the event that a bus error occurs during a burst refill, any subsequent cache block refills are discontinued.

The TX19 processor core recognizes bus error signaling during bus cycles of its own; thus when a write buffer unit is used to write data to external memory, the processor never takes a Bus Error

exception. In that case, external hardware must suspend the erroneous bus operation by delivering the interrupt signal.

When a bus error occurs during a load, the contents of the processor's destination register is set to X.

9.1.10 System Call Exception

Cause

This exception occurs when a SYSCALL instruction is executed.

Handling

Figure 9-8 highlights the CP0 register fields that are used to handle this exception.

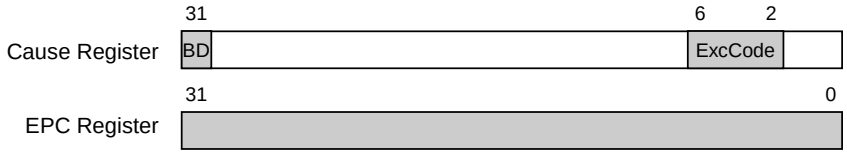


Figure 9-8 System Call Exception

1. The Sys code (8) is set into the ExcCode field in the Cause register.
2. The EPC register stores the program counter on the exception. If the processor is executing an instruction 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 in the EPC register saves the ISA mode that was in effect prior to the exception.
3. Processor context in the Status register is stacked, and the KUc and IEc bits are cleared to enter Kernel mode and disable all interrupts (see 9.1.5, *Saving and Restoring Processor Context*).
4. The processor jumps to the exception handler located at address 0x8000_0080.

When a System Call exception occurs, control is transferred to an exception handler. The unused bits (bits 25-6) in a SYSCALL instruction are 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

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

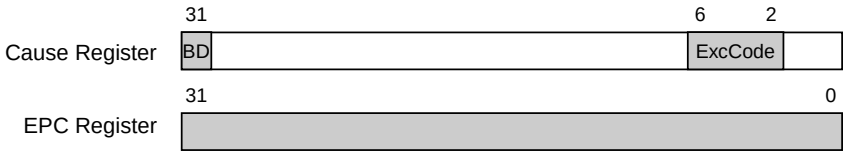


Figure 9-9 Breakpoint Exception

1. The Bp code (9) is set into the ExcCode field in the Cause register.
2. The EPC register stores the program counter on the exception. If the processor is executing an instruction 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 in the EPC register saves the ISA mode that was in effect prior to the exception.
3. Processor context in the Status register is stacked, and the KUc and IEc bits are cleared to enter Kernel mode and disable all interrupts (see 9.1.5, *Saving and Restoring Processor Context*).
4. The processor jumps to the exception handler located at address 0x8000_0080.

When a Breakpoint exception occurs, control is transferred to an exception handler. The unused bits (bits 25-6 in the 32-bit instruction, bits 10-5 in the 16-bit instruction) in a BREAK instruction are 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 (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) or a Special instruction with an undefined minor opcode (bits 5-0)
- execute an unimplemented instruction (LWCz, SWCz)

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

- execute an instruction with an undefined instruction code 1110_1xxx_yyy0_1001, 1110_1xxx_yyy1_0001, 1110_1xxx_yyy1_0101, 1100_100i_iiii_iiii or 0110_0110_iiii_iiii
- execute an unimplemented instruction (LWU, LD, SD, DADDU, DSUBU, DADDIU, DMULT, DMULTU, DDIV, DDIVU, DSLL, DSRL, DSRA, DSLLV, DSRLV, DSRAV)
- EXTEND an instruction that can not be EXTENDed

Handling

Figure 9-10 highlights the CP0 register fields that are used to handle this exception.

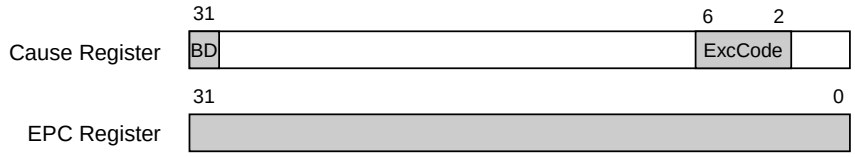


Figure 9-10 Reserved Instruction Exception

1. The RI code (10) is set into the ExcCode field in the Cause register.
2. The EPC register stores the program counter on the exception. If the processor is executing an instruction 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 in the EPC register saves the ISA mode that was in effect prior to the exception.
3. Processor context in the Status register is stacked, and the KUc and IEc bits are cleared to enter Kernel mode and disable all interrupts (see 9.1.5, *Saving and Restoring Processor Context*).
4. If the exception occurs while the processor was in 16-bit ISA mode, the processor switches to 32-bit ISA mode.
5. The processor jumps to the exception handler located at address 0x8000_0080.

The TX19 performs direct segment mapping of virtual to physical addresses; it does not have an on-chip table lookaside buffer (TLB). If TLB instructions are encountered, the processor turns them into NOPs (No Operations) instead of generating a Reserved Instruction exception.

9.1.13 Coprocessor Unusable Exception

Cause

This exception occurs when an attempt is made to:

- execute a coprocessor instruction when the corresponding coprocessor unit is marked unusable in the CU[*z*] bit in the Status register (where *z* is the coprocessor unit number, 0 to 3)
- execute a CP0 instruction in User mode when the CU[0] bit in the Status register is cleared

The coprocessor instructions, LWC*z*, SWC*z*, MTC*z*, MFC*z*, CTC*z*, CFC*z*, COP*z*, BC*z*T, BC*z*F, BC*z*TL and BC*z*FL, and the system control coprocessor (CP0) instructions, MTC0, MFC0, RFE and COP0, can generate this exception.

Kernel-mode execution of CP0 instructions never causes this exception regardless of the setting of the CU[0] bit in the Status register.

Handling

Figure 9-11 highlights the CP0 register fields that are used to handle this exception.

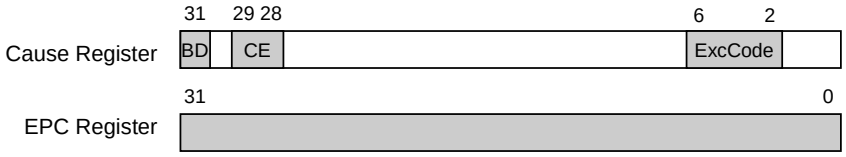


Figure 9-11 Coprocessor Unusable Exception

1. The CpU code (11) is set into the ExcCode field in the Cause register.
2. The CE field in the Cause register shows which of the four coprocessor units was referenced when an exception occurred.
3. The EPC register stores the program counter on the exception. If the processor is executing an instruction 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 in the EPC register saves the ISA mode that was in effect prior to the exception.
4. Processor context in the Status register is stacked, and the KUc and IEc bits are cleared to enter Kernel mode and disable all interrupts (see 9.1.5, *Saving and Restoring Processor Context*).
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 0x8000_0080.

9.1.14 Integer Overflow Exception

Cause

This exception occurs when the ADD, ADDI or SUB instruction results in two's-complement overflow.

Handling

Figure 9-12 highlights the CP0 register fields that are used to handle this exception.

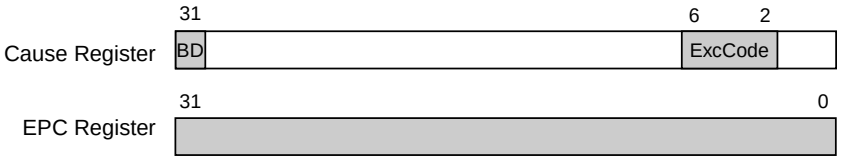


Figure 9-12 Integer Overflow Exception

1. The Ov code (12) is set into the ExcCode field in the Cause register.
2. The EPC register stores the program counter on the exception. If the processor is executing an instruction 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 in the EPC register saves the ISA mode that was in effect prior to the exception.
3. Processor context in the Status register is stacked, and the KUc and IEc bits are cleared to enter Kernel mode and disable all interrupts (see 9.1.5, *Saving and Restoring Processor Context*).
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 0x8000_0080.

9.1.15 Reset Exception

Cause

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

Handling

1. All the CP0 registers are initialized as shown in Chapter 8.
2. The processor jumps to the exception handler located at address 0XBFC0_0000.

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

9.2 Interrupts

The TX19 provides a nonmaskable 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 TX19 recognizes a nonmaskable interrupt, 7 levels of maskable hardware interrupts and 4 maskable software interrupts. Interrupt exceptions are processed by hardware and then serviced by software (interrupt service routines). See 9.1.6, *Maskable Interrupt Exception*, and 9.1.7, *Nonmaskable Interrupt Exception*, for how interrupt exceptions are handled by processor hardware.

Sources of nonmaskable interrupts can be an assertion of the processor's NMI* input or on-chip peripherals such as watchdog timers. See individual hardware user's manuals for possible on-chip sources of nonmaskable interrupts. Nonmaskable interrupts are for implementation of critical interrupt routines and can not be masked (disabled) by software; they are always recognized 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. Typically, they are submitted to the interrupt controller, which then turns them into a 3-bit priority level for input to the TX19 processor core. The processor compares its current interrupt mask level (i.e., the CMask[2:0] field in the Status register) with the interrupt request priority to determine whether to service the interrupt immediately or to delay service. The interrupt is serviced immediately if its priority is higher than the mask level. The mask level is updated during interrupt recognition.

There are four software interrupts, Swi0 to Swi3. 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.

The Current Interrupt Enable bit, IEc, in the Status register globally controls the enabling of all maskable interrupts.

9.2.2 Maskable Interrupt Priorities

The TX19 allows a priority model to be built for maskable interrupts. The processor's maskable interrupt port is 3-bit wide, allowing eight levels of hardware interrupts to be defined, from the highest 7 (binary 111) to the lowest 0 (binary 000). A priority-0 interrupt would never successfully stop execution of a program of any priority. Interrupt priorities for various interrupt sources are to be defined by the interrupt mode control register within the interrupt controller.

Software interrupts Swi0 to Swi2 have a priority level of 1, and Swi3 has a priority level of 4. Although Swi0, Swi1 and Swi2 have an equal priority level, Swi2 has a higher priority than Swi1 and Swi1 has a higher priority than Swi0 when more than one software interrupt requests are made simultaneously.

9.2.3 Maskable Interrupt Vectors

The four software interrupts are vectored to distinct service routines as shown in Table 9-2, *Exception Vector Addresses*. When a hardware interrupt occurs, the processor jumps to the default address (0x8000_0160); the interrupt service routine must then check the interrupt controller in order to determine the source of the interrupt, read the corresponding vector address and transfer control to it.

9.2.4 Maskable Interrupt Recognition

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

- Interrupts are enabled (The IEc bit in the Status register is set).
- The interrupt request priority is higher than the current mask level set in the CMask[2:0] field in the Status register.
- If the interrupt is software-generated, the corresponding mask bit (SwiMask[3:0]) in the Status register is cleared.

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

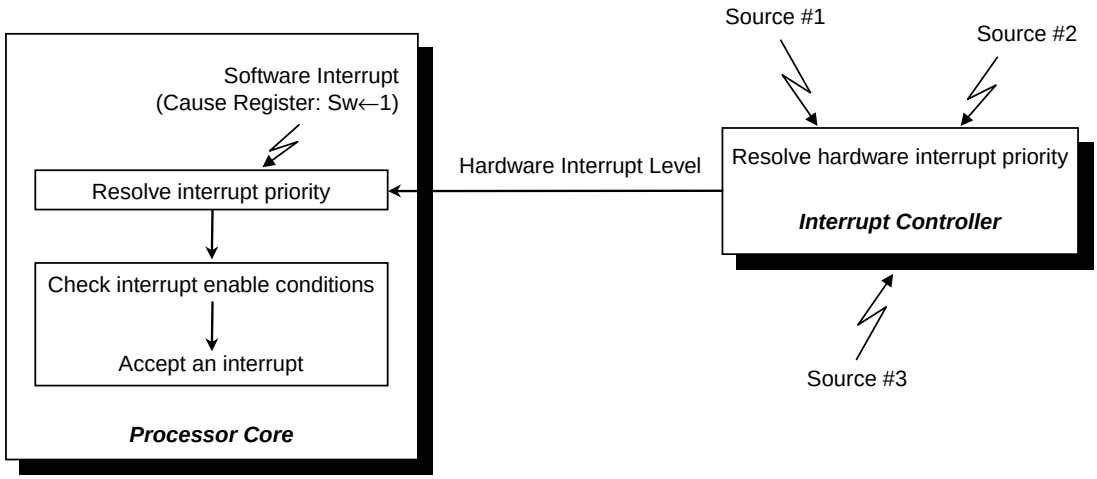


Figure 9-13 Maskable Interrupt Recognition

9.2.5 Interrupt Mask Level

Whenever the processor accepts an interrupt, it automatically saves its request level in the Interrupt Mask field, CMask[2:0], in the Status register. This allows all equal- and lower-priority interrupts to be left pending while the interrupt is being serviced. If the interrupt is software-generated, the mask is saved immediately on interrupt recognition. If the interrupt is hardware-generated, the mask is saved at the time the processor reads out its interrupt vector.

The processor continuously compares the processor's mask level to the priorities of requested interrupts. Thus, before the writing of the CMask[2:0] field, the processor can accept a higher-priority interrupt.

The Status register has a two-level stack for the interrupt mask level. When the processor accepts an interrupt, the contents of the Current Interrupt Mask field, CMask[2:0], is saved to the Previous Interrupt Mask field, PMask[2:0]. Returning from an interrupt routine is made through the Restore From Exception (RFE) instruction. When the RFE instruction is executed at completion of an interrupt service routine, the mask level is restored to what it was before the interrupt was recognized. This is done by popping the PMask[2:0] value to CMask[2:0].

When the processor takes an interrupt exception, it automatically clears the IEc (Interrupt Enable, Current) bit to turn off all interrupts. Once the mask level for the current interrupt is set, the IEc bit can be changed to allow higher-priority interrupts.

9.3 Debug Exceptions

There are Single-step and Debug Breakpoint exceptions in the TX19. 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 TX19 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 9-14.

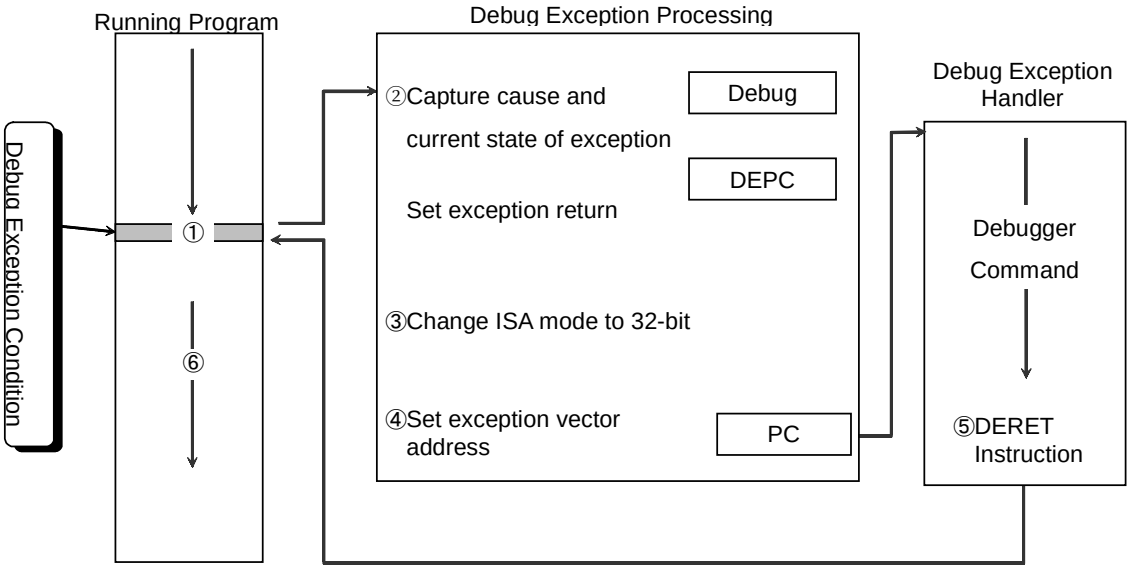


Figure 9-14 Exception Operation

1. The currently executing instruction and any subsequent instructions in the pipeline are aborted.
2. 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 was in effect when the exception occurred.

3. The processor enters Kernel mode and turns off 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 9-4 gives the types of debug exceptions that can occur in the TX19 processor.

Table 9-4 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, and 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. The operation of the SDBBP instruction is undefined if a debug exception is being serviced (i.e., the DM bit in the Debug register is set).

9.3.3 Debug Exception Priorities

Single-step and Debug Breakpoint exceptions do not occur at the same time; the Single-step exception has higher priority than the Debug Breakpoint exception.

A debug exception and a general exception may occur simultaneously. In that case, the processor first services the debug exception; however, at this point, the Status, Cause, EPC and/or BadVAddr registers are updated with information about the pending general exception. Additionally, the NIS or OES bit in the Debug register is set to indicate that a Nonmaskable Interrupt exception or another general exception occurred.

Debug and general exceptions should be serviced in the sequence shown in Figure 9-15.

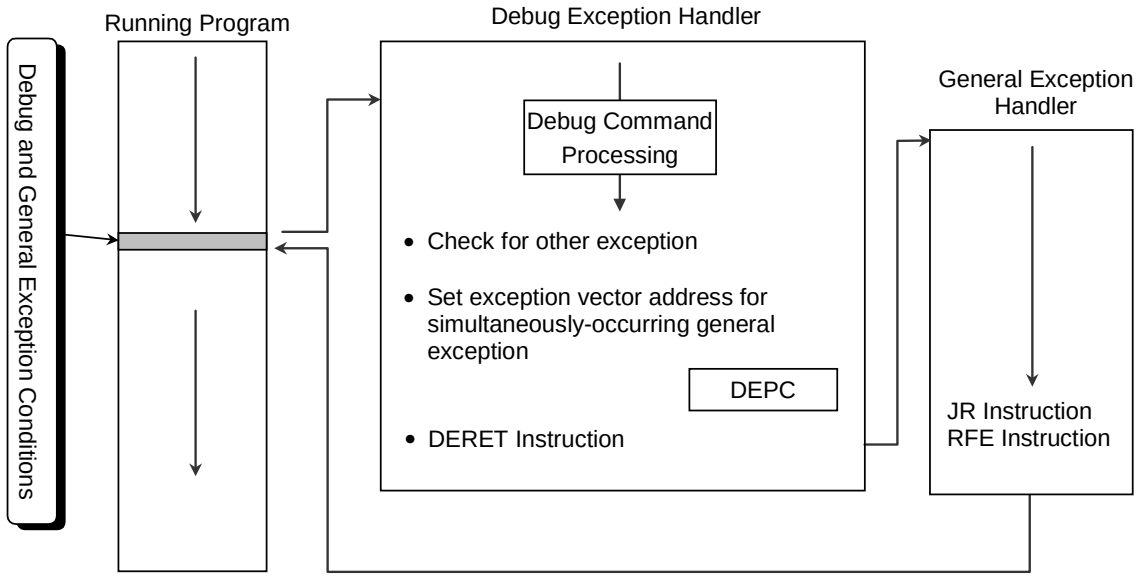


Figure 9-15 Debug Exception Priorities

On a debug exception, the DEPC register saves the address of the exception-causing instruction. So that a simultaneously-occurring general exception will be serviced after completion of the debug exception processing, the debug exception handler must check the Debug and Cause registers to determine which type of general exception occurred, if any, and loads the DEPC register with the exception vector address accordingly. This way, execution of the DERET instruction at the end of the debug exception handler directly transfers control to the general exception handler.

A Single-step exception may coincide with an Address Error exception during an instruction fetch, but not with any other type of general exceptions. In cases where an Address Error exception occurs during an instruction fetch, that instruction is never executed; so a Debug Breakpoint exception is not generated at the same time.

Table 9-5 gives the general exception vector address that should be loaded into the DEPC register by the debug exception handler.

Table 9-5 General Exception Vector Addresses

Debug Register		Cause Register			Simultaneous General Exception	Exception Vector (Required DEPC Register Value)
NIS	OES	ExcCode	IL[2:0]	Sw[3:0]		
1	0	x	x	x	Nonmaskable Interrupt	0xBFC0_0000
0	1	≠0	x	x	Other than Reset, Nonmaskable Interrupt or Maskable Interrupt	0x8000_0080 (BEV=0) 0xBFC0_0180 (BEV=1)
			≥4	x	Hardware Interrupt	0x8000_0160 (BEV=0) 0xBFC0_0260 (BEV=1)
		=0	1-3	1xxx	Software Interrupt Swi3	0x8000_0140 (BEV=0) 0xBFC0_0240 (BEV=1)
				0xxx	Hardware Interrupt	0x8000_0160 (BEV=0) 0xBFC0_0260 (BEV=1)
				1xxx	Software Interrupt Swi3	0x8000_0140 (BEV=0) 0xBFC0_0240 (BEV=1)
			0	01xx	Software Interrupt Swi2	0x8000_0130 (BEV=0) 0xBFC0_0230 (BEV=1)
				001x	Software Interrupt Swi1	0x8000_0120 (BEV=0) 0xBFC0_0220 (BEV=1)
				0001	Software Interrupt Swi0	0x8000_0110 (BEV=0) 0xBFC0_0210 (BEV=1)

Note: x signifies a "don't care."

9.3.4 Exception Masking

While a debug exception is being serviced, the processor masks all the other exceptions. This is accomplished as follows:

- When a Bus Error event occurs, the BsF bit in the Debug register is set to flag its occurrence.
- All maskable interrupts are turned off while a debug exception is being serviced. (Maskable interrupts are unmasked by the execution of a DERET instruction.)
- A nonmaskable interrupt is left pending until a return from a debug exception is made through the DERET instruction.
- The processor operation is undefined if any other exception occurs during debug exception processing.

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) bit in the Debug register.
3. Gets out of the forced "Kernel mode, interrupt-disabled" state and makes the Status register's KUc and IEc bits valid again.

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 9-16 highlights the CP0 register fields that are used to handle this exception.

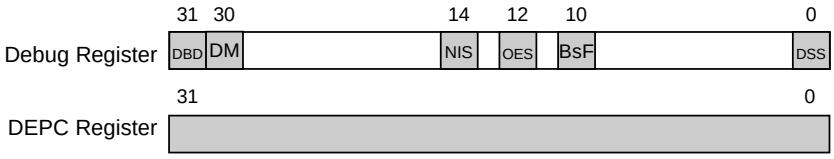


Figure 9-16 Single-step Exception

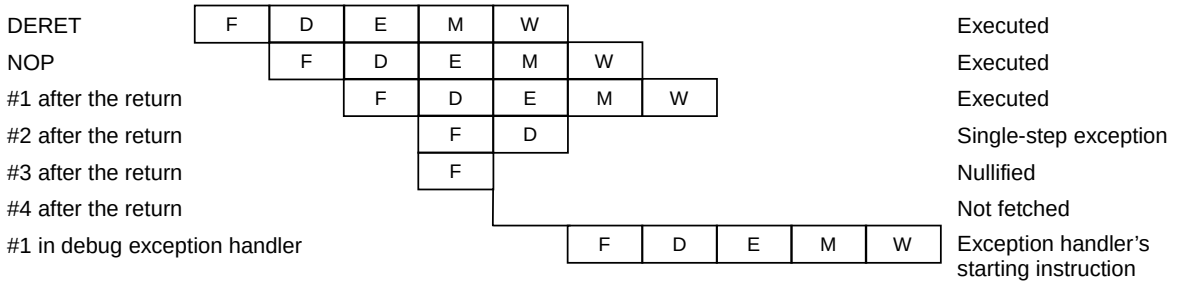
1. The DM and DSS bits in the Debug register are set. In the event that a general exception event occurred simultaneously, the NIS or OES bit is set. That a Single-step exception occurred means the SSt bit had been set.
2. The DEPC register stores the program counter on the exception. The least-significant bit in the DEPC register saves the ISA mode that was in effect prior to the exception.
3. The processor enters Kernel mode and turns off all interrupts, independent of the setting of the Status register.
4. The processor jumps to the exception handler located at address 0xBFC0_0200.

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

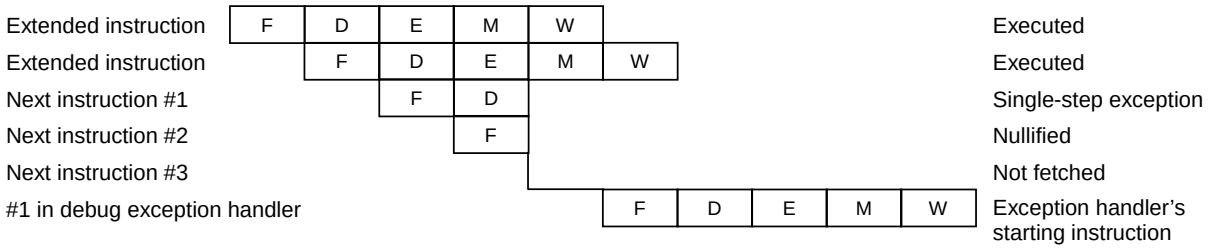
Figure 9-17)

- a debug exception is being serviced (i.e., the DM bit in the Debug register is set)
- the instruction immediately following an EXTENDed instruction (see Figure 9-18)



The DEPC register points at instruction #2 after the return from the exception.

Figure 9-17 CPU Pipeline Operation After the DERET Instruction



The DEPC register saves the address of next instruction #1.

Figure 9-18 CPU Pipeline Operation After an EXTENDed Instruction

9.3.8 Debug Breakpoint Exception

Cause

This exception occurs when an SDBBP instruction is executed.

Handling

Figure 9-19 highlights the CP0 register fields that are used to handle this exception.

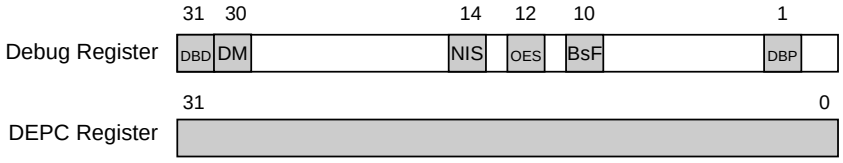


Figure 9-19 Debug Breakpoint Exception

1. The DM and DBP bits in the Debug register are set. In the event that a general exception event occurred simultaneously, the NIS or OES bit is 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 turns off all interrupts, independent of the setting 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_0200.

The unused bits (bits 25-6 in the 32-bit ISA, bits 10-5 in the 16-bit ISA) in an SDBBP instruction are available for use as software parameters to pass additional information 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 SDBBP instruction is not re-executed. If the SDBBP 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 TX19 provides hardware support for many levels of power reduction. The Halt and Doze modes are invoked by register programming, and the Reduced Frequency mode is invoked by a cooperation between register programming and a clock generator. This chapter describes the power management features and capabilities provided by the TX19.

10.1 Power-Saving Modes

Figure 10-1 illustrates the power-saving modes provided by the TX19.

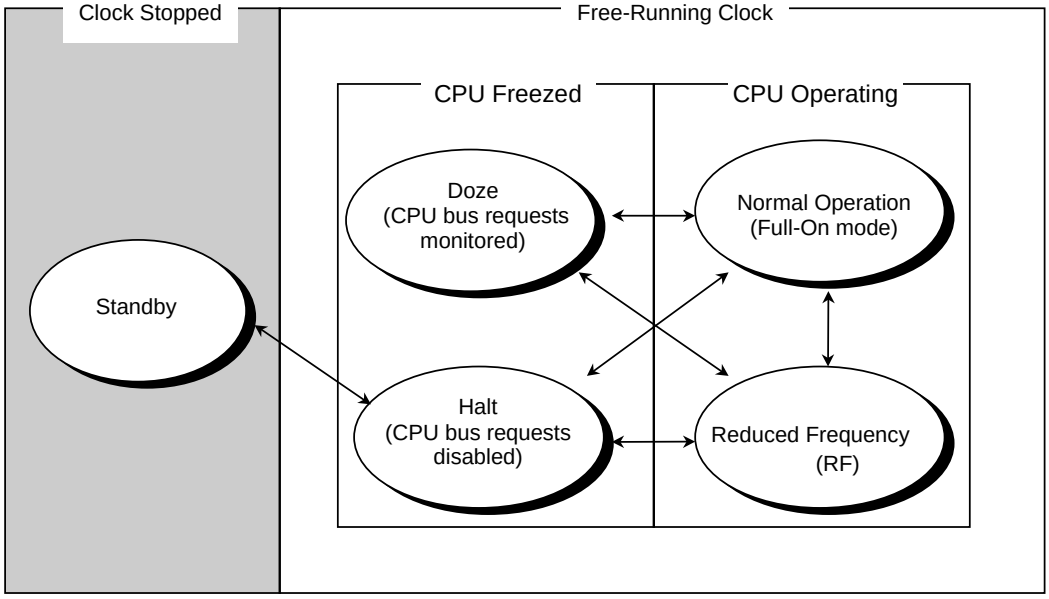


Figure 10-1 Power-Saving Modes

The TX19 has many methods of dynamically controlling power consumption during operation. Table 10-1 describes the available power-saving modes.

Table 10-1 Power-Saving Modes

Mode	Description
Standby Mode	For lowest power operation, the processor clock can be removed altogerher. 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 TX19 processor assumes bus mastership. Halt mode can be entered by programming the Config register.
Doze Mode	In Doze mode, all activities of the processor stop except for the CPU bus monitor, which continues to operate and recognizes bus requests. Doze mode can be entered by programming the Config register.
Reduced Frequency (RF) Mode	The processor clock can be programmed to run at fc/2, fc/4 or fc/8 to reduce power consumption, where fc is the full-speed frequency of the processor. RF mode can be entered by programming the Config register.
Normal Mode (Full-On Mode)	This is the default power state of the TX19 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 modes, see respective hardware user's manuals.

10.2 Halt Mode

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

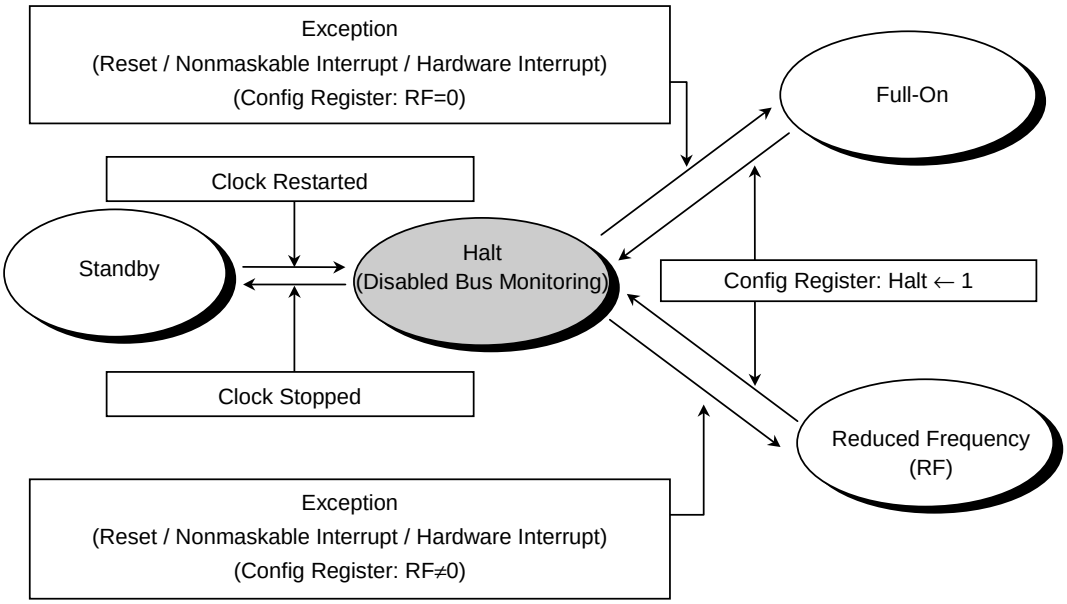


Figure 10-2 Halt Mode

Halt mode freezes the "processor core," preserving the pipeline state. In Halt mode, the processor ignores any external bus requests, so 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.

The processor enters Halt mode when software writes a 1 to the Halt bit in the Config register while in Full-On or RF mode. A wakeup from Halt mode can be achieved by causing a Reset, Nonmaskable Interrupt or Maskable Hardware Interrupt exception. Any of such exceptions causes clearing of the Halt bit, followed by processing of that 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 initiates a wakeup.

10.3 Doze Mode

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

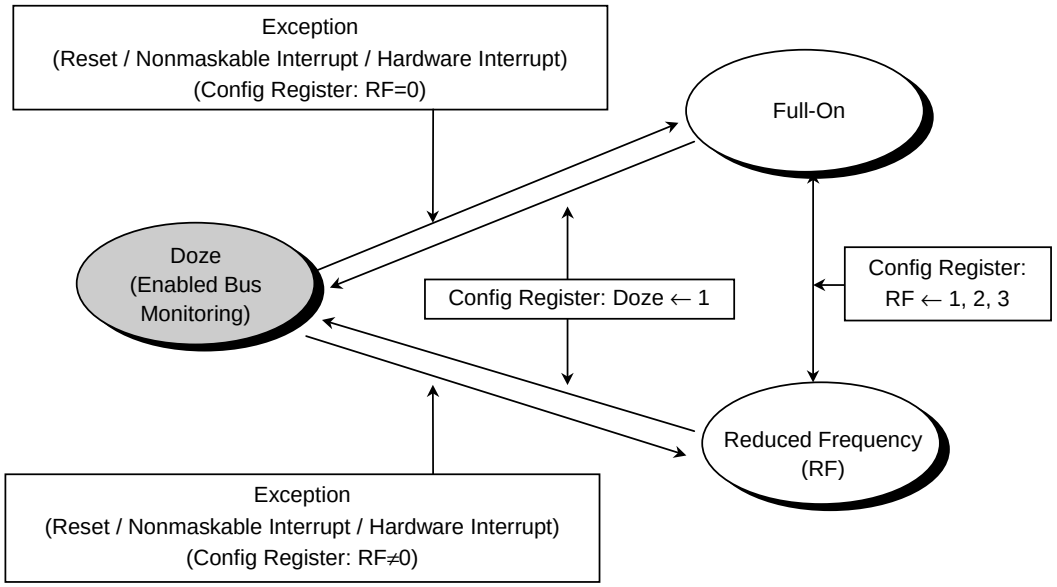


Figure 10-3 Doze Mode

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.

The processor enters Doze mode when software writes a 1 to the Doze bit in the Config register while in Full-On or RF mode. A wakeup from Doze mode can be achieved by causing a Reset, Nonmaskable Interrupt or Maskable Hardware Interrupt exception. Any of such exceptions causes clearing of the Doze bit, followed by processing of that 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.

10.4 Reduced Frequency (RF) Mode

The processor clock can be programmed to run at fc/2, fc/4 or fc/8 to reduce power consumption, where fc is the full-speed frequency of the processor. The division is by a power-of-2, as programmed in the RF[1:0] bits in the Config register. The value of the RF[1:0] field in the Config register is driven to the processor output, which in turn is used as input to the on-chip clock generator to indicate the clock divisor. The processor is brought back to full speed by resetting the RF[1:0] bits to zero.

If the Halt or Doze bit in the Config register is set while the processor is operating in RF mode, the processor enter Halt or Doze mode accordingly. A Reset, Nonmaskable Interrupt or Maskable Hardware Interrupt exception brings the processor back into RF 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***.

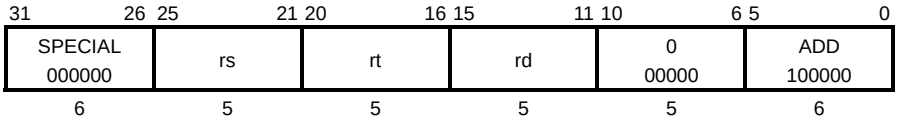
ADD *rd*, *rs*, *rt*

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*.

An Integer Overflow exception is taken on 2's-complement overflow, which occurs if the signs of the addends are the same and the sign of the sum is different. 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.

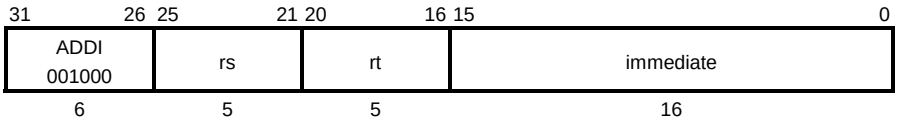
ADDI *rt, rs, immediate*

Add Immediate

Operation

rt ⇐ *rs* + *immediate*

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.

The *immediate* field is 16 bits in length. This gives a range of -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.

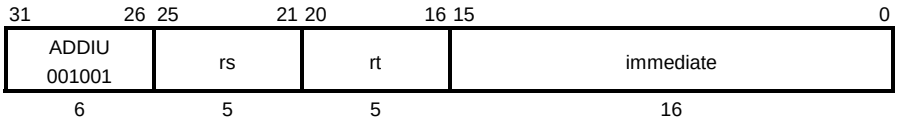
ADDIU *rt*, *rs*, *immediate*

Add Immediate Unsigned

Operation

rt ⇐ *rs* + *immediate*

Instruction Encoding



Description

Although the opcode stands for "Add Immediate Unsigned," 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

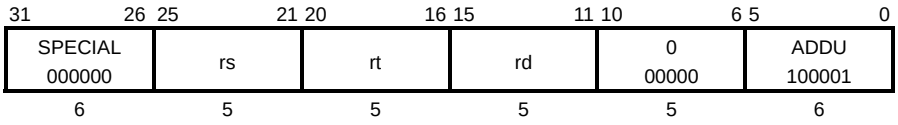
ADDU *rd*, *rs*, *rt*

Add Unsigned

Operation

rd ⇐ *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

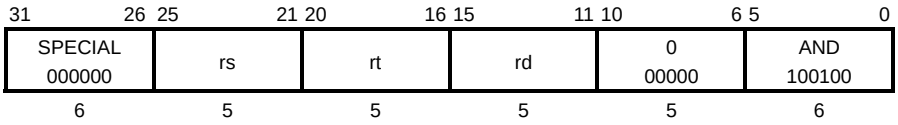
AND *rd*, *rs*, *rt*

AND

Operation

rd ⇐ *rs* AND *rt*

Instruction Encoding



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

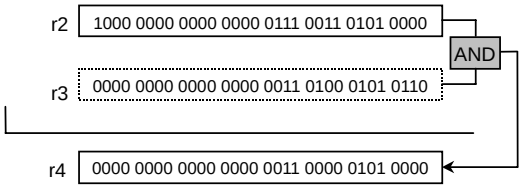
None

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.



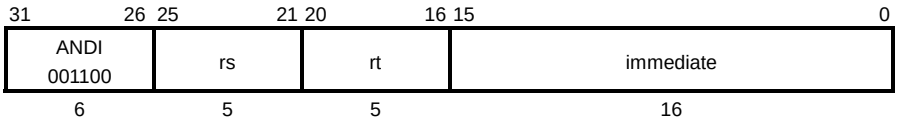
ANDI *rt*, *rs*, *immediate*

Logical AND Immediate

Operation

rt ⇐ *rs* AND *immediate*

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

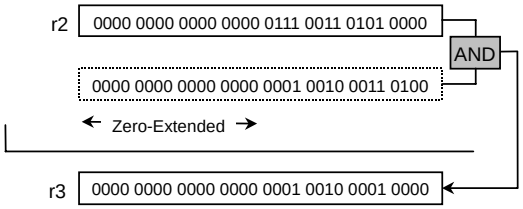
None

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.



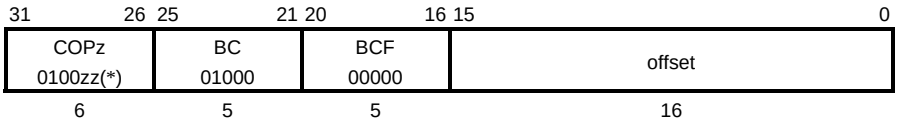
BC*z*F *offset*

Branch On Coprocessor z False

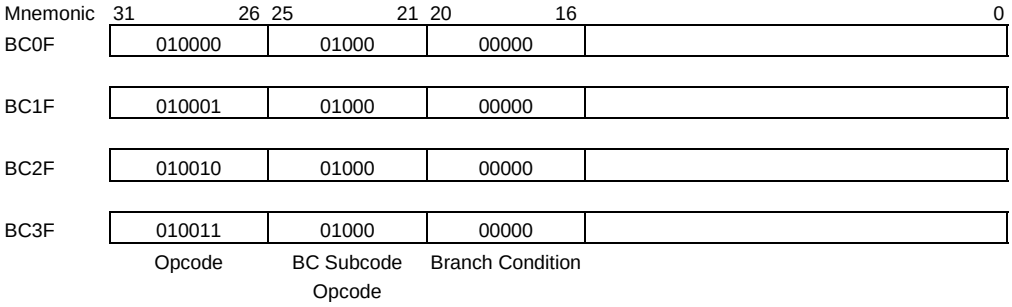
Operation

if coprocessor *z*'s condition signal is false
then pc ⇐ pc + *offset*

Instruction Encoding



The following shows the opcode bit encoding. The two low-order bits in the opcode field signify the coprocessor unit number.



Description

If the coprocessor unit *z*'s condition signal (CPCOND), as sampled during execution of the previous instruction, is false, then the program branches to the target address with a delay of one instruction (i.e., two instruction 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. If the coprocessor unit *z*'s condition signal (CPCOND) is true, the program just continues to the next instruction.

Exceptions

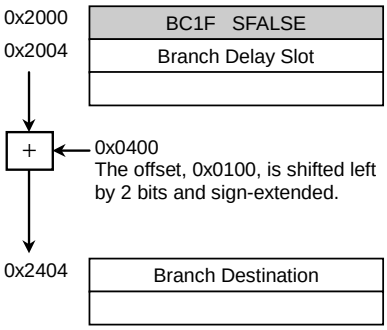
Coprocessor Unusable exception

Example

```
BC1F SFALSE
```

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

If the coprocessor unit 1's condition signal (CPCOND) is false, the processor transfers program control to address 0x2404. The branch takes effect after the instruction in the branch delay slot is executed.



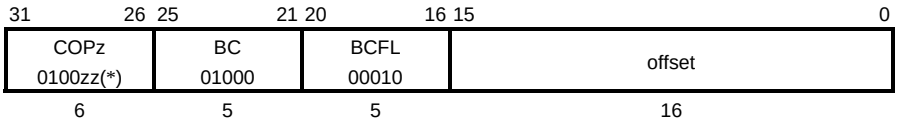
BC*z*FL *offset*

Branch On Coprocessor z False Likely

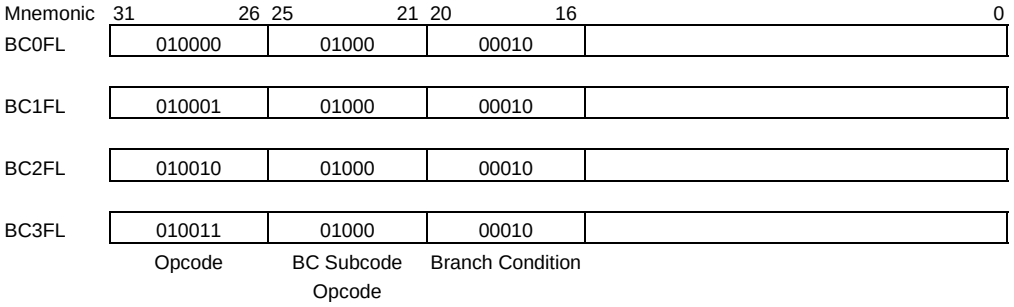
Operation

if coprocessor *z*'s condition signal is false
then pc ⇐ pc + *offset*

Instruction Encoding



The following shows the opcode bit encoding. The two low-order bits in the opcode field signify the coprocessor unit number.



Description

If the coprocessor unit *z*'s condition signal (CPCOND), as sampled during execution of the previous instruction, is false, then the program branches to the target address with a delay of one instruction (i.e., two instruction 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. If the coprocessor unit *z*'s condition signal (CPCOND) is true, the instruction in the branch delay slot is nullified.

Exceptions

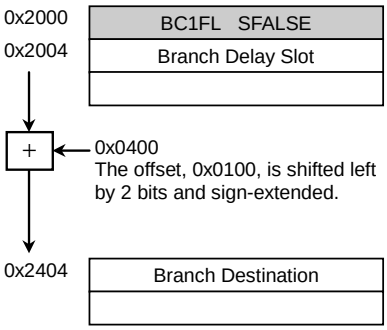
Coprocessor Unusable exception

Example

```
BC1FL SFALSE
```

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

If the coprocessor unit 1's condition signal (CPCOND) is false, 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 slot is nullified.



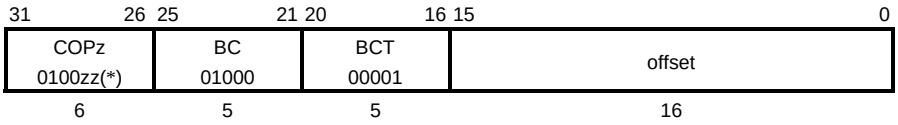
BCzT *offset*

Branch On Coprocessor z True

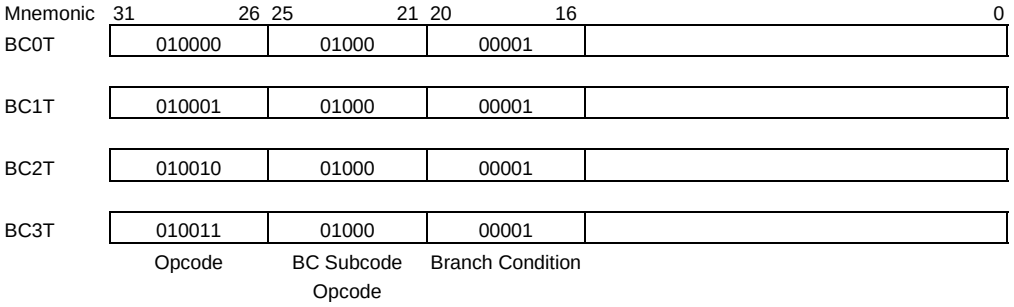
Operation

if coprocessor *z*'s condition signal is true
then pc ⇐ pc + *offset*

Instruction Encoding



The following shows the opcode bit encoding. The two low-order bits in the opcode field signify the coprocessor unit number.



Description

If the coprocessor unit *z*'s condition signal (CPCOND), as sampled during execution of the previous instruction, is true, then the program branches to the target address with a delay of one instruction (i.e., two instruction 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. If the coprocessor unit *z*'s condition signal (CPCOND) is false, the program just continues to the next instruction.

Exceptions

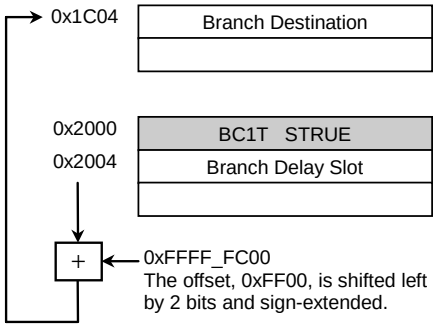
Coprocessor Unusable exception

Example

```
BC1T STRUE
```

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

If the coprocessor unit 1's condition signal (CPCOND) is true, the processor transfers program control to address 0x1C04. The branch takes effect after the instruction in the branch delay slot is executed.



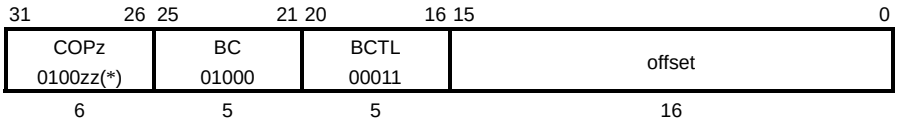
BC*z*TL *offset*

Branch On Coprocessor z True Likely

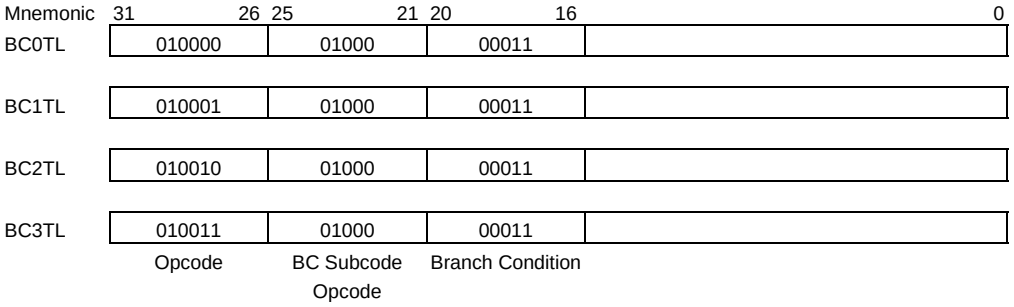
Operation

if coprocessor *z*'s condition signal is true
then pc ⇐ pc + *offset*

Instruction Encoding



The following shows the opcode bit encoding. The two low-order bits in the opcode field signify the coprocessor unit number.



Description

If the coprocessor unit *z*'s condition signal (CPCOND), as sampled during execution of the previous instruction, is true, then the program branches to the target address with a delay of one instruction (i.e., two instruction 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. If the coprocessor unit *z*'s condition signal (CPCOND) is false, the instruction in the branch delay slot is nullified.

Exceptions

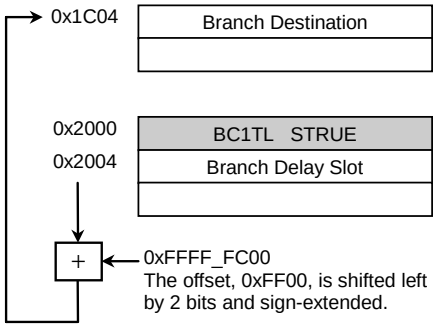
Coprocessor Unusable exception

Example

```
BC1TL STRUE
```

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

If the coprocessor unit 1's condition signal (CPCOND) is true, 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.



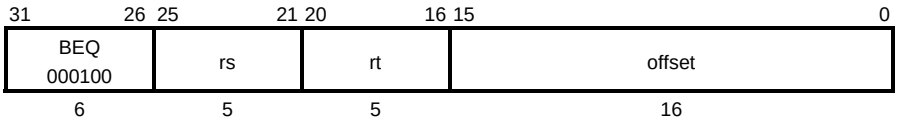
BEQ *rs*, *rt*, *offset*

Branch On Equal

Operation

if *rs* = *rt* then pc ⇐ pc + *offset*

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 (i.e., two instruction 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

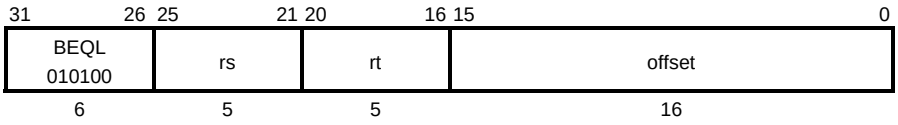
BEQL *rs*, *rt*, *offset*

Branch On Equal Likely

Operation

if *rs* = *rt* then pc ⇐ pc + *offset*

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 (i.e., two instruction cycles). If the branch is not taken, the instruction in the branch delay slot 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

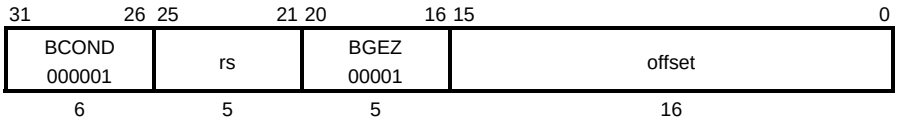
BGEZ *rs*, *offset*

Branch On Greater Than Or Equal To Zero

Operation

if *rs* ≥ 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction 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

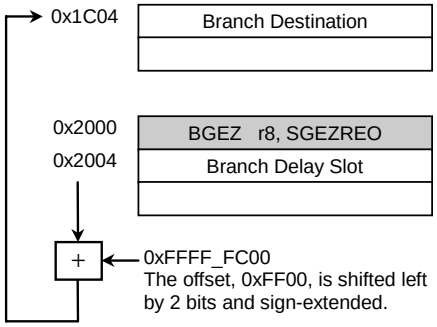
None

Example

```
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 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, SGEZREO
0x2004
Branch Delay Slot
+
0xFFFF_FC00
The offset, 0xFF00, is shifted left
by 2 bits and sign-extended.

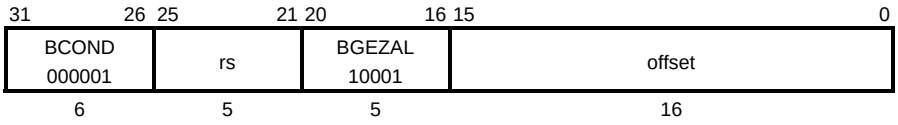
BGEZAL *rs*, *offset*

Branch On Greater Than or Equal To Zero And Link

Operation

r31 ⇐ pc +8; if *rs* ≥ 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction cycles), and saves the address of the instruction following the branch delay slot (PC+8) in the link register, r31. 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 is not restartable, 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

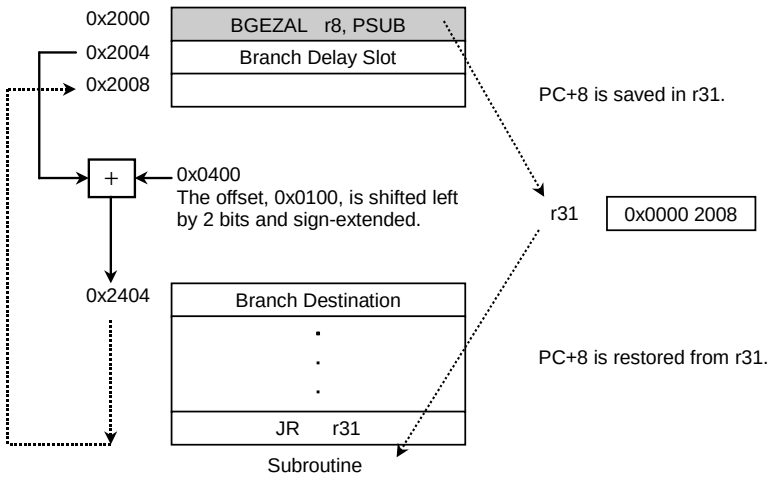
```
BGEZAL r8,PSUB
```

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 relative offset 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
```


0x2000
0x2004
0x2008
BGEZAL r8, PSUB
Branch Delay Slot
PC+8 is saved in r31.
+
0x0400
The offset, 0x0100, is shifted left
by 2 bits and sign-extended.
r31
0x0000 2008
0x2404
Branch Destination
PC+8 is restored from r31.
JR r31
Subroutine

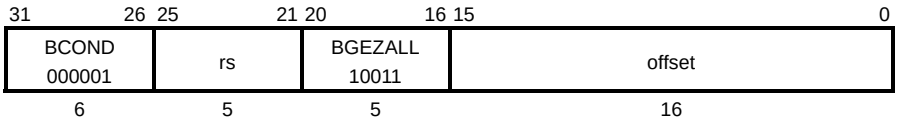
BGEZALL *rs*, *offset*

Branch On Greater Than Or Equal To Zero And Link Likely

Operation

r31 ⇐ pc +8; if *rs* ≥ 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction cycles), and saves the address of the instruction following the branch delay slot (PC+8) in the link register, r31. If the branch is not taken, the instruction in the branch delay slot 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 is not restartable, 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

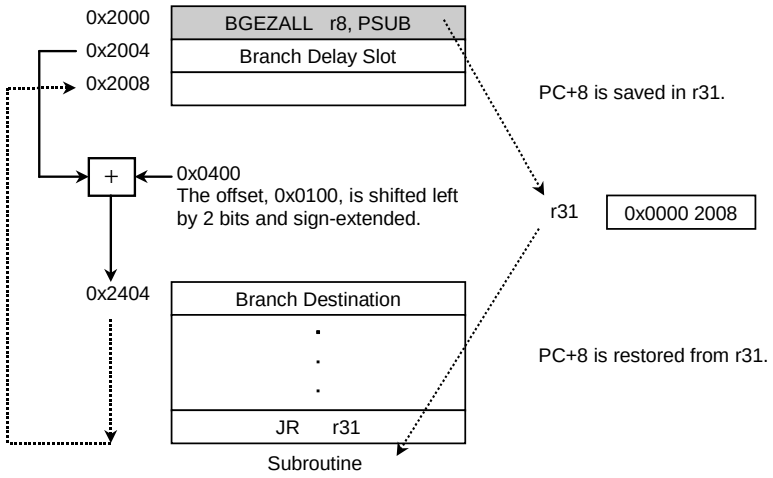
```
BGEZALL r8,PSUB
```

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 relative offset 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 not 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
```



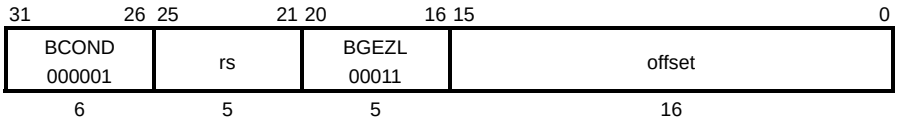
BGEZL *rs*, *offset*

Branch On Greater Than Or Equal To Zero Likely

Operation

if *rs* ≥ 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction cycles). If the branch is not taken, the instruction in the branch delay slot is nullified and the program continues to the next instruction. 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

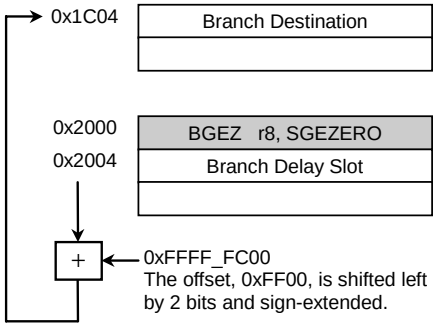
None

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 relative offset 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.

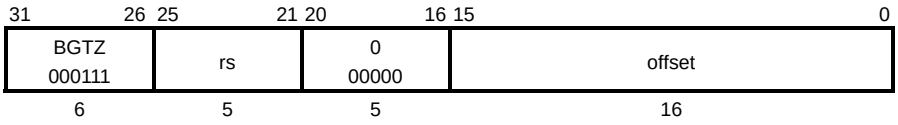
BGTZ *rs*, *offset*

Branch On Greater Than Zero

Operation

if *rs* > 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction 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

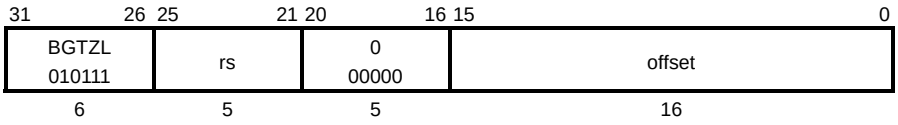
BGTZL *rs*, *offset*

Branch On Greater Than Zero Likely

Operation

if *rs* > 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction cycles). If the branch is not taken, the instruction in the branch delay slot 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

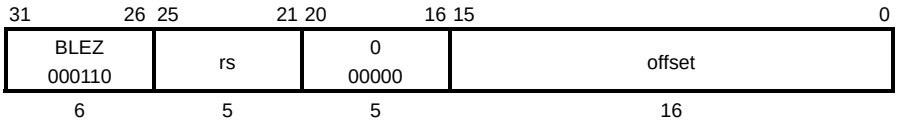
BLEZ *rs*, *offset*

Branch On Less Than Or Equal To Zero

Operation

if *rs* ≤ 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction 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

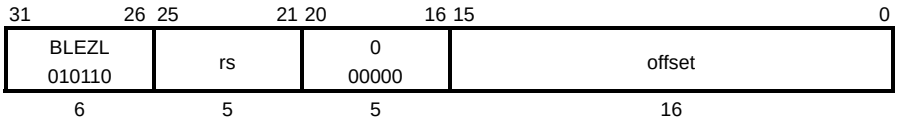
BLEZL *rs*, *offset*

Branch On Less Than Or Equal To Zero Likely

Operation

if *rs* ≤ 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction cycles). If the branch is not taken, the instruction in the branch delay slot 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

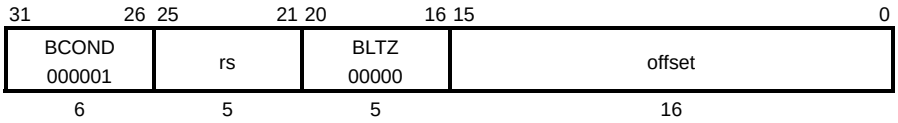
BLTZ *rs*, *offset*

Branch On Less Than Zero

Operation

if *rs* < 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction 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

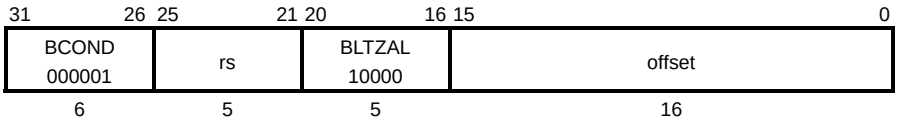
BLTZAL *rs*, *offset*

Branch On Less Than Zero And Link

Operation

r31 ⇐ pc +8; if *rs* < 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction 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. 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 restartable, 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

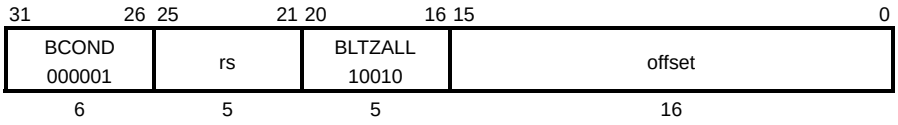
BLTZALL *rs*, *offset*

Branch On Less Than Zero And Link Likely

Operation

r31 ⇐ pc +8; if *rs* < 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction cycles), and saves the address of the instruction following the branch delay slot (PC+8) in the link register, r31. If the branch is not taken, the instruction in the branch delay slot 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 is not restartable, 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

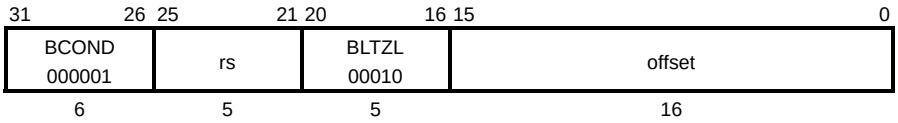
BLTZL *rs*, *offset*

Branch On Less Than Zero Likely

Operation

if *rs* < 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction cycles). If the branch is not taken, the instruction in the branch delay slot 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

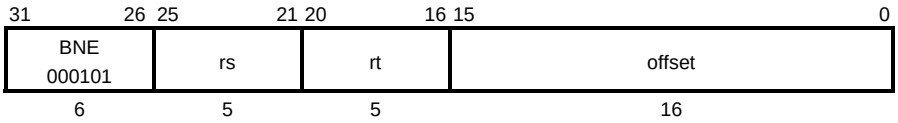
BNE *rs*, *rt*, *offset*

Branch On Not Equal

Operation

if *rs* ≠ *rt* then pc ⇐ pc + *offset*

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 (i.e., two instruction 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

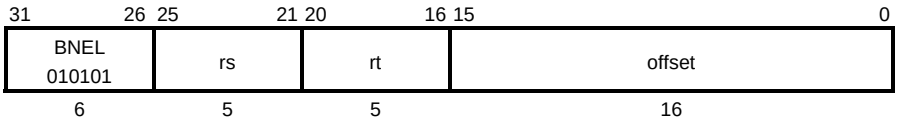
BNEL *rs*, *rt*, *offset*

Branch On Not Equal Likely

Operation

if *rs* ≠ *rt* then pc ⇐ pc + *offset*

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 (i.e., two instruction cycles). If the branch is not taken, the instruction in the branch delay slot 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

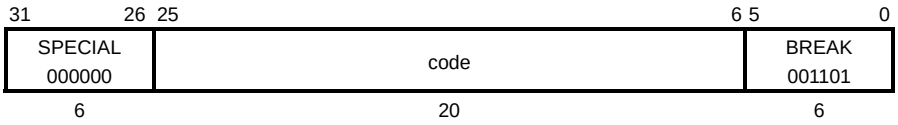
BREAK *code*

Breakpoint Exception

Operation

Breakpoint exception

Instruction Encoding



Description

When this instruction is executed, a breakpoint trap 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

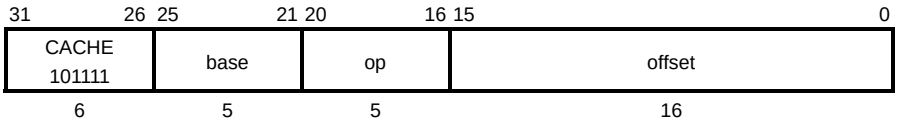
CACHE *op*, *offset* (*base*)

Cache Operation

Operation

Cache operation

Instruction Encoding

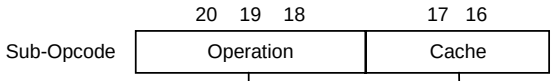


Description

The 16-bit immediate *offset* is sign-extended and added to the contents of general-purpose register *base* to form a virtual address. The virtual address is translated to a physical address. The 5-bit sub-opcode (bits 20-16) specifies a cache operation for that address.

Attempts by a User-mode program to execute the CACHE instruction when the CU[0] bit in the Status register is cleared causes a Coprocessor Unusable exception. Kernel-mode programs can always execute the CACHE instruction. The operation of this instruction is undefined if cache is not available.

Bits 20 to 18 and bits 17-16 of the instruction specify the operation and cache as follows.



Code[17:16]	Name	Cache
00	I	Instruction
01	D	Data
1x	-	Reserved

Code[20:18]	Code[17:16]	Name	Operation
000	00/01	Index Invalidate	Clears the Valid bit in all tags for the index specified by the physical address, irrespective of a cache hit or a cache miss. This operation is valid only when the cache is marked "disabled" in the ICE or DCE bit of the Config register.
001	00/01	Index LRU Bit Clear	Clears the LRU bit in all tags for the index specified by the physical address.
010	00/01	Index Lock Bit Clear	Clears the Lock bit in all tags for the index specified by the physical address.
100	00/01	Hit Invalidate	In the case of a cache hit, clears the Valid bit of only the matching tag for the index.

Exceptions

Coprocessor Unusable exception

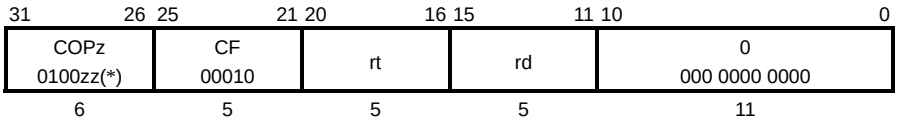
CFC*z* *rt*, *rd*

Move Control From Coprocessor z

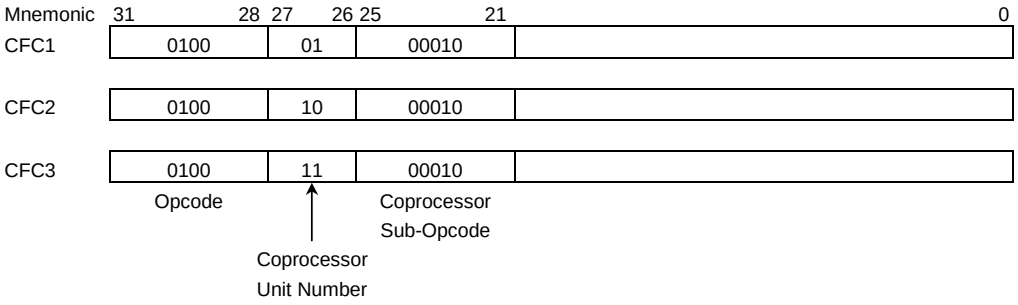
Operation

rt ⇐ coprocessor control register *rd* of coprocessor unit *z*

Instruction Encoding



The following shows the opcode bit encoding. The two low-order bits in the opcode field signify the coprocessor unit number.



Description

The contents of coprocessor control register *rd* of coprocessor unit *z* is loaded into general-purpose register *rt*.

This instruction is not valid for CP0.

Exceptions

Coprocessor Unusable exception

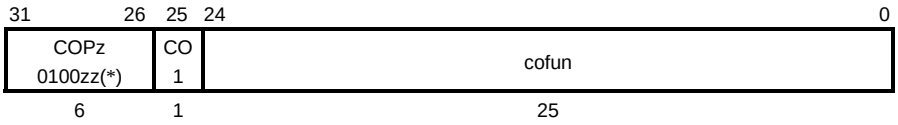
COP*z cofun*

Coprocessor z Operation

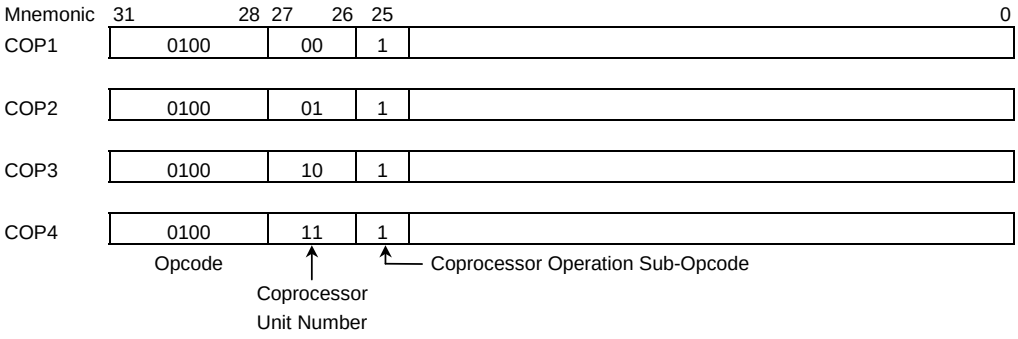
Operation

Coprocessor operation (*z*, *cofun*)

Instruction Encoding



The following shows the opcode bit encoding. The two low-order bits in the opcode field signify the coprocessor unit number.



Description

A coprocessor operation specified by *cofun* is performed on coprocessor unit *z*.

The operation may specify or reference internal coprocessor registers and may change the state of the coprocessor condition signal (CPCOND), but does not alter the internal state of the processor or the cache/memory system.

Exceptions

Coprocessor Unusable exception

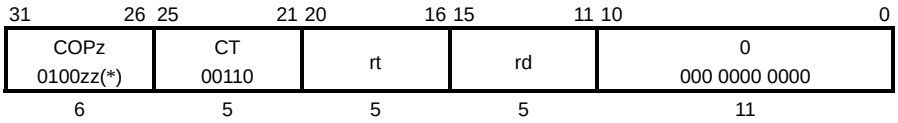
CTC*z rt*, *rd*

Move Control To Coprocessor z

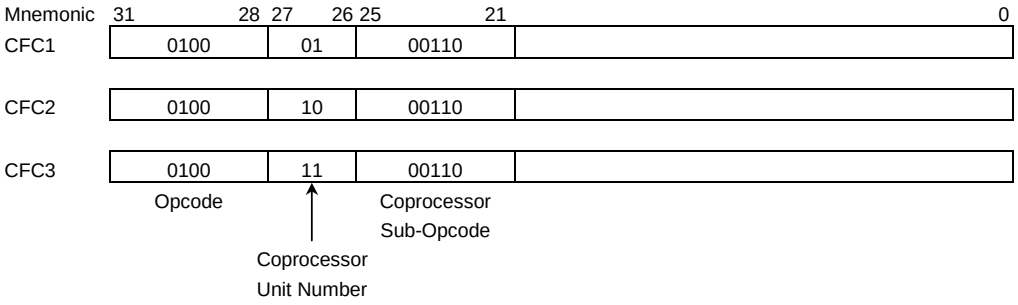
Operation

Coprocessor control register *rd* of coprocessor unit *z* ⇐ *rt*

Instruction Encoding



The following shows the opcode bit encoding. The two low-order bits in the opcode field signify the coprocessor unit number.



Description

The contents of general-purpose register *rt* is loaded into coprocessor control register *rd* of coprocessor unit *z*.

This instruction is not valid for CP0.

Exceptions

Coprocessor Unusable exception

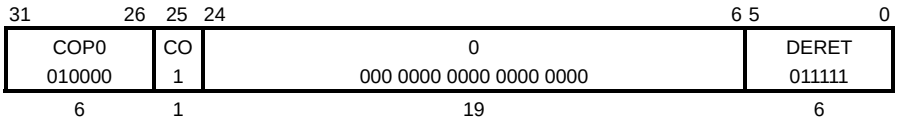
DERET

Debug Exception Return

Operation

pc ⇐ DEPC

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.

Like branch instructions, the DERET instruction has a branch delay slot and is executed with a delay of one instruction (i.e., two instruction 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 NOP instruction must be inserted in the delay slot following the DERET instruction. Also, the DERET instruction may not be in a jump or branch delay slot.

The operation of the DERET instruction is undefined if the processor is not in a 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. It is strictly prohibited to execute a DERET instruction immediately after the MTC0 instruction that writes to the Debug register. Otherwise, the contents of the Debug register would become undefined. Additionally, it is strictly prohibited to execute a DERET instruction immediately after the MFC0 instruction that reads from the Debug register. Otherwise, the contents of the Debug register would become undefined.

Exceptions

Coprocessor Unusable exception

DIV *rs*, *rt*

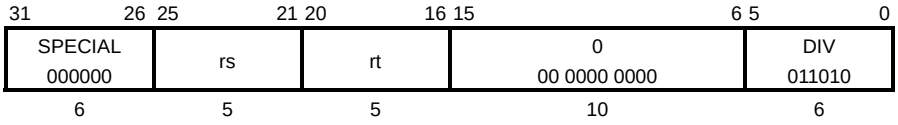
Divide

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*. 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 integer overflow exceptions.

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 cache misses, delay cycles and exceptions occur.

If the DIV instruction is followed by an MFHI, MFLO, MADD or MADDU 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*

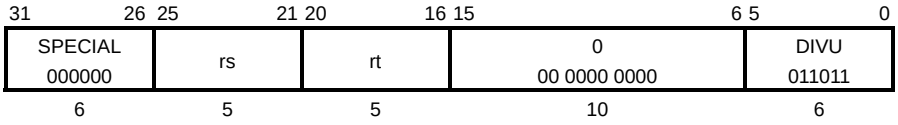
Divide Unsigned

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 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

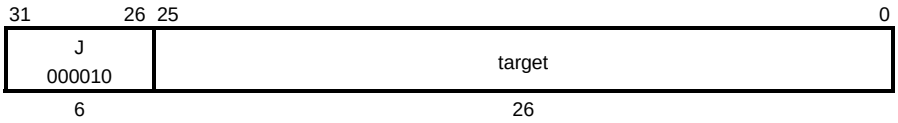
J *target*

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 (i.e., two instruction 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

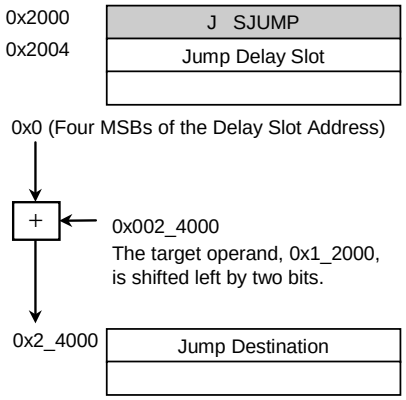
None

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

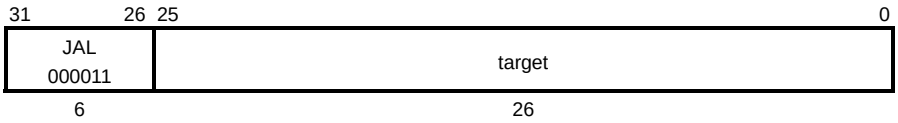
JAL *target*

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 (i.e., two instruction 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 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

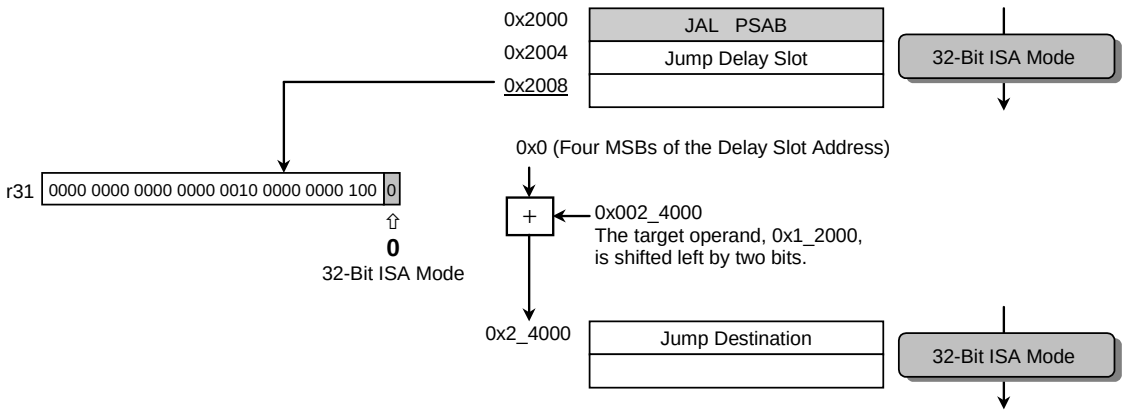
None

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
0x2004
0x2008
JAL PSAB
Jump Delay Slot
32-Bit ISA Mode
r31
0000 0000 0000 0000 0010 0000 0000 100
0
0
32-Bit 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
32-Bit ISA Mode

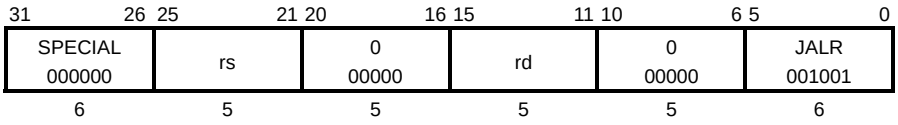
JALR (*rd*,) *rs*

Jump And Link Register

Operation

rd or r31 ⇐ pc + 8; 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 (i.e., two instruction cycles). The least-significant bit of *rs* is interpreted as the ISA mode specifier. The address of the instruction after the jump delay slot is saved in general-purpose register *rd*. If *rd* is omitted, the default is r31 (ra).

Register *rd* may not be the same one as register *rs* because such an instruction is not restartable, 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 these low-order bits are not zero, an Address Error exception will occur when the processor fetches the instruction at the jump destination.

Exceptions

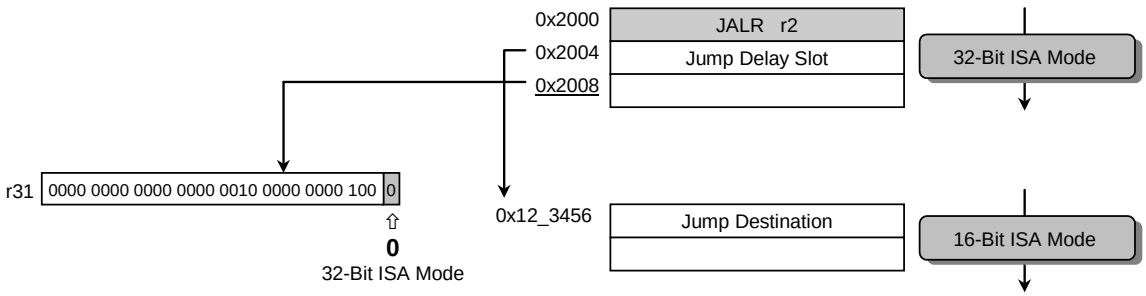
None

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, together with the ISA mode bit.


0x2000
JALR r2
0x2004
Jump Delay Slot
0x2008
32-Bit ISA Mode
r31
0000 0000 0000 0000 0010 0000 0000 100
0
0
32-Bit ISA Mode
0x12_3456
Jump Destination
16-Bit ISA Mode

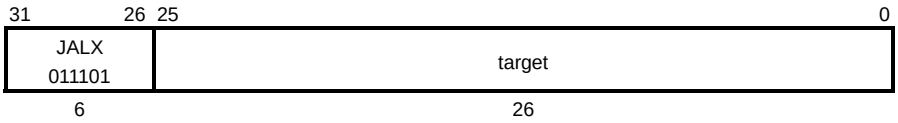
JALX *target*

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 (i.e., two instruction 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 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

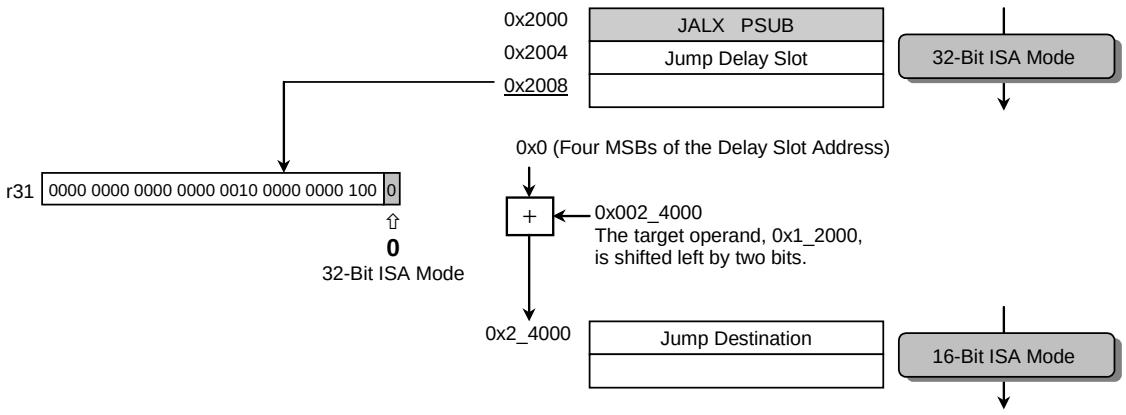
None

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, together with the ISA mode bit.


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

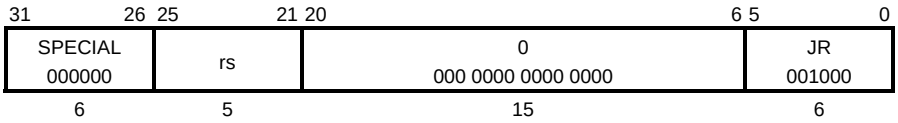
JR *rs*

Jump Register

Operation

pc ⇐ *rs*

Instruction Code



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 (i.e., two instruction cycles). The least-significant bit of *rs* is interpreted as the ISA mode specifier.

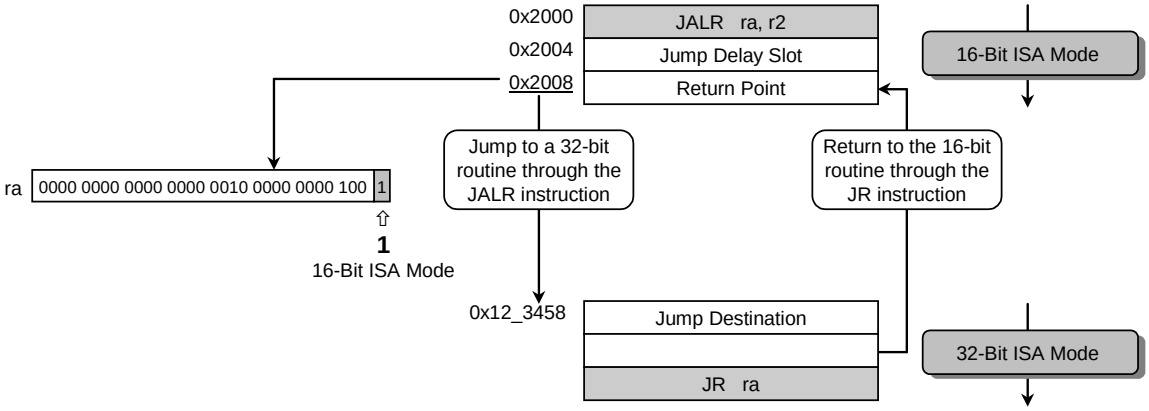
In 32-bit ISA mode, all instructions must be aligned on word boundaries. Therefore, when jumping to a 32-bit routine, the two low-order bits of the target register (*rs*) must be zero. If these low-order bits are not zero, an Address Error exception will occur when the processor fetches the instruction 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
0x2004
0x2008
JALR ra, r2
Jump Delay Slot
Return Point
16-Bit ISA Mode
Jump to a 32-bit routine through the JALR instruction
Return to the 16-bit routine through the JR instruction
ra
0000 0000 0000 0000 0010 0000 0000 100
1
1
16-Bit ISA Mode
0x12_3458
Jump Destination
JR ra
32-Bit ISA Mode

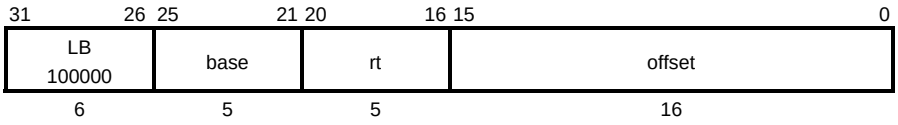
LB *rt*, *offset* (*base*)

Load Byte

Operation

rt ⇐ {*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

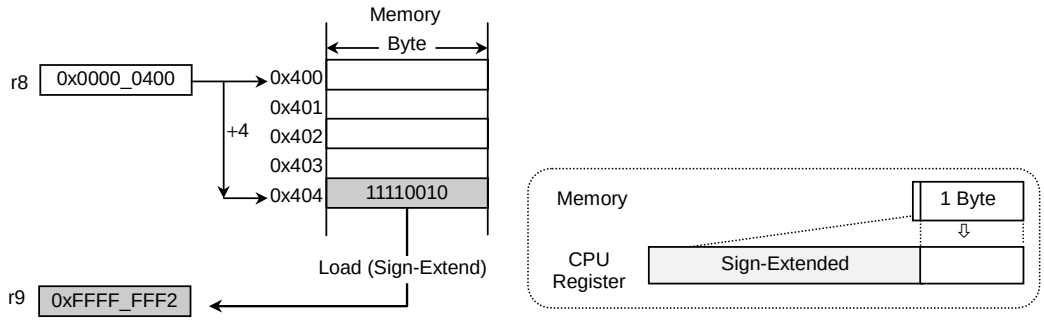
Address Error exception

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.



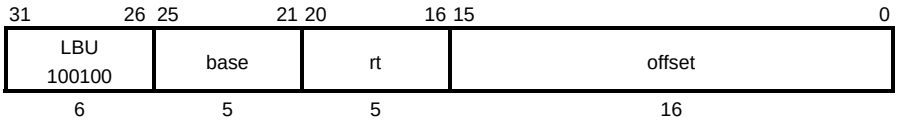
LBU *rt*, *offset* (*base*)

Load Byte Unsigned

Operation

rt ⇐ {*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

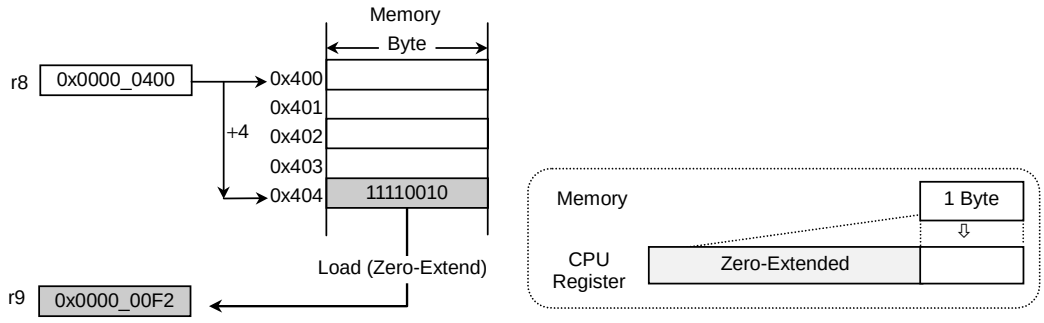
Address Error exception

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.



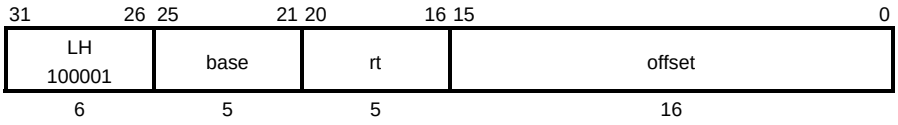
LH *rt*, *offset* (*base*)

Load Halfword

Operation

rt ⇐ {*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:

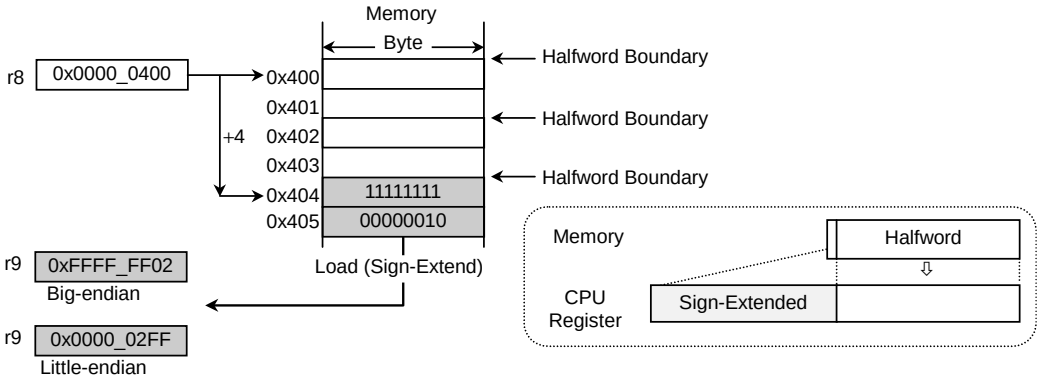
```
LH r9,4(r8)
```

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
r8
0x0000_0400
0x400
0x401
0x402
0x403
0x404
0x405
+4
11111111
00000010
Halfword Boundary
Halfword Boundary
Halfword Boundary
Load (Sign-Extend)
r9
0xFFFF_FF02
Big-endian
r9
0x0000_02FF
Little-endian
Memory
Halfword
CPU
Register
Sign-Extended

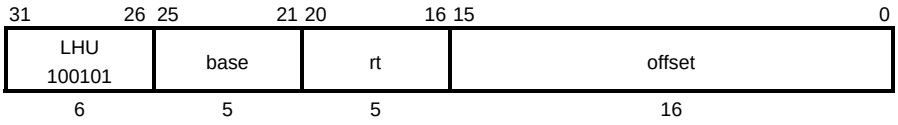
LHU *rt*, *offset* (*base*)

Load Halfword Unsigned

Operation

rt ⇐ {*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:

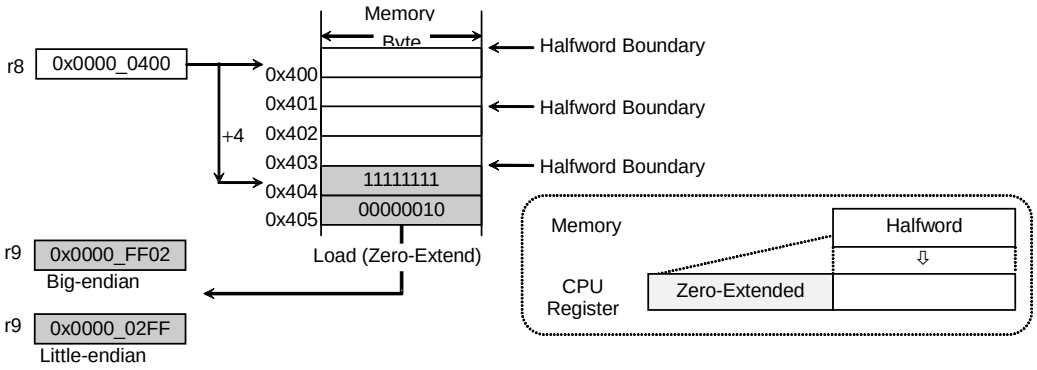
```
LHU r9,4(r8)
```

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
r8
0x0000_0400
0x400
0x401
0x402
0x403
0x404
0x405
+4
11111111
00000010
Halfword Boundary
Halfword Boundary
Halfword Boundary
Load (Zero-Extend)
r9
0x0000_FF02
Big-endian
r9
0x0000_02FF
Little-endian
Memory
Halfword
CPU
Register
Zero-Extended

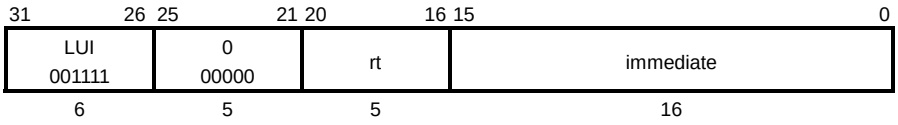
LUI *rt*, *immediate*

Load Upper Immediate

Operation

rt ⇐ *immediate* || 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.

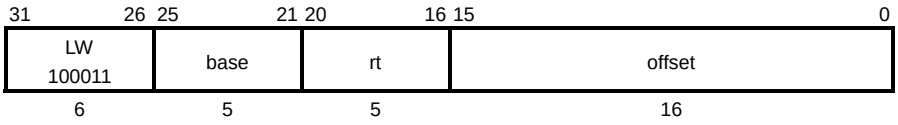
LW *rt*, *offset* (*base*)

Load Word

Operation

rt ⇐ {*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 are 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:

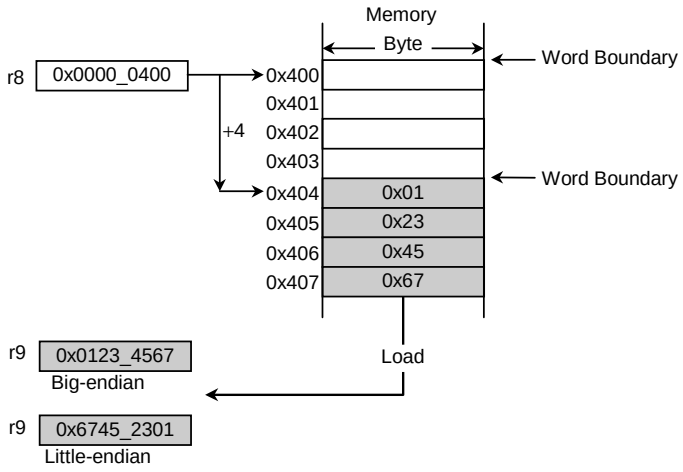
```
LW r9,4(r8)
```

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
0x402
0x403
+4
Word Boundary
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
Load
r9
0x0123_4567
Big-endian
r9
0x6745_2301
Little-endian

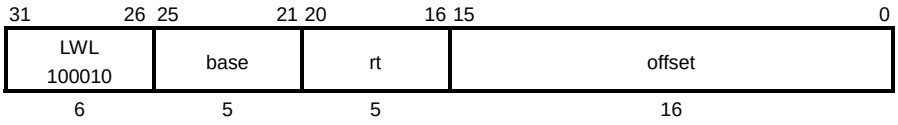
LWL *rt*, *offset* (*base*)

Load Word Left

Operation

rt ⇐ {*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.

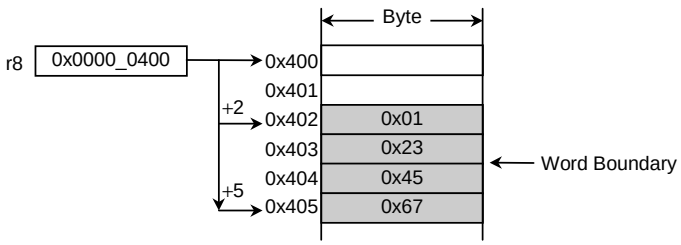
The LWL and LWR instructions are used in combination to load a misaligned word from memory into a general-purpose register.

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.

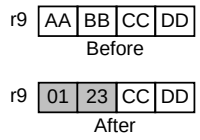


- Big-endian mode

  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.



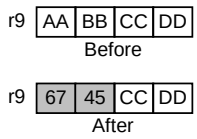
(a) Big-endian

- 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

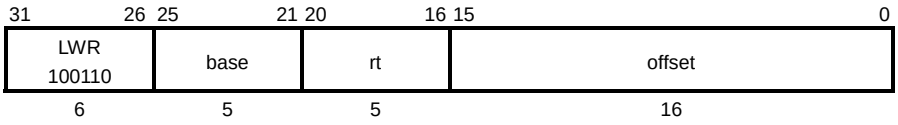
LWR *rt*, *offset* (*base*)

Load Word Right

Operation

rt ⇐ {*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.

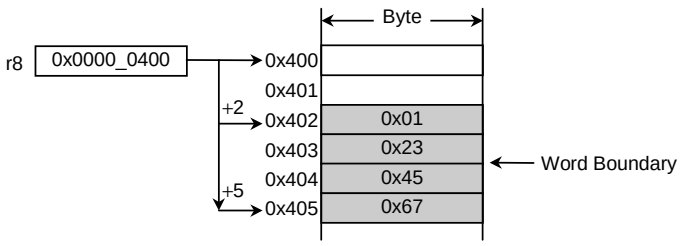
The LWL and LWR instructions are used in combination to load a misaligned word from memory into a general-purpose register.

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.

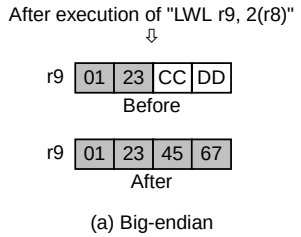


- Big-endian mode

  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.



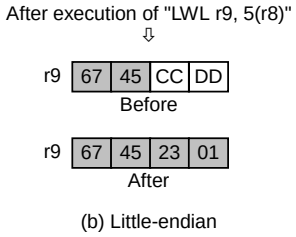
(a) Big-endian

- 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

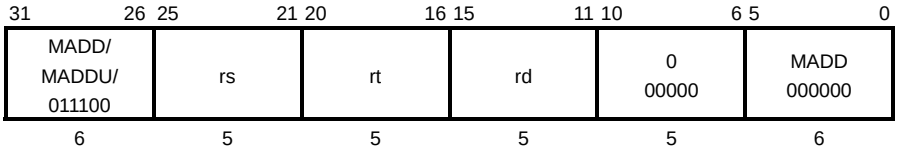
Operation

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, causing the copy of the low-order word into a general-purpose register to be discarded.

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 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

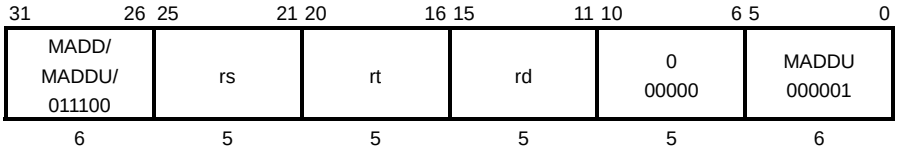
Operation

HI ⇐ 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 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, causing the copy of the low-order word into a general-purpose register to be discarded.

No integer overflow exception occurs under any circumstances.

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.

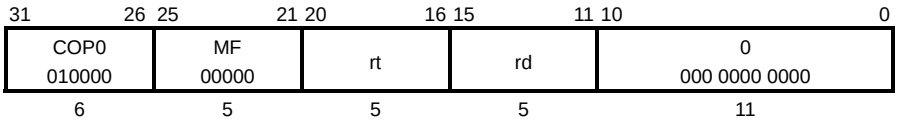
MFC0 *rt*, *rd*

Move From System Control Coprocessor (CP0)

Operation

rt ⇐ coprocessor register *rd* of CP0

Instruction Encoding



Description

The contents of CP0 register *rd* is loaded into general-purpose register *rt*.

The MFC0 instruction may not attempt to read the contents of the Status register immediately before the RFE instruction. Otherwise, the contents of the Status register become undefined.

Likewise, the MFC0 instruction may not attempt to read the contents of the Debug register immediately before the DERET instruction. Otherwise, the contents of the Debug register become undefined.

Exceptions

Coprocessor Unusable exception

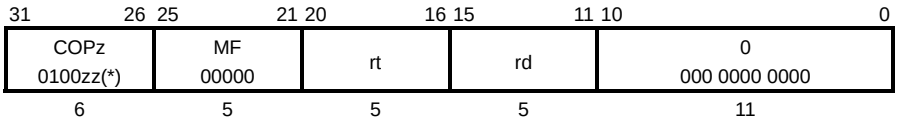
MFC*z* *rt*, *rd*

Move From Coprocessor z

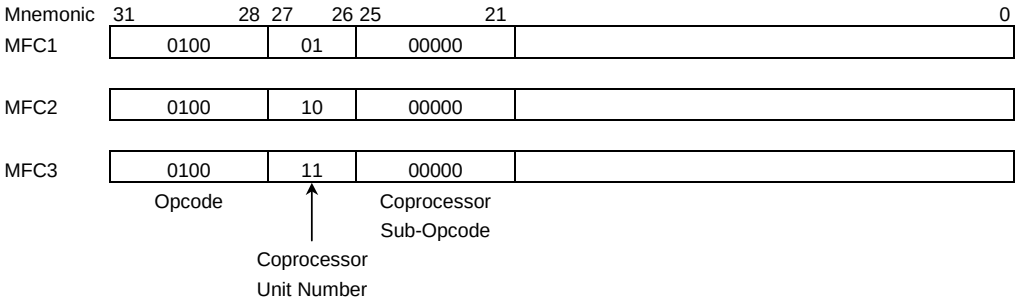
Operation

rt ⇐ coprocessor register *rd* of coprocessor unit *z*

Instruction Encoding



The following shows the opcode bit encoding. The two low-order bits in the opcode field signify the coprocessor unit number.



Description

The contents of coprocessor register *rd* of coprocessor unit *z* is loaded into general-purpose register *rt*.

Exceptions

Coprocessor Unusable exception

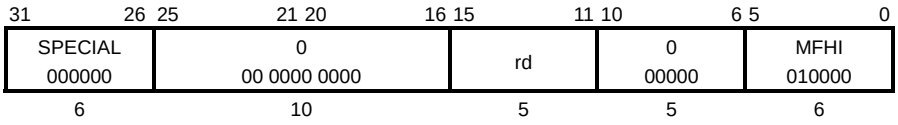
MFHI *rd*

Move From HI

Operation

rd ⇐ HI

Instruction Encoding



Description

The contents of the HI register is loaded into general-purpose register *rd*.

Exceptions

None

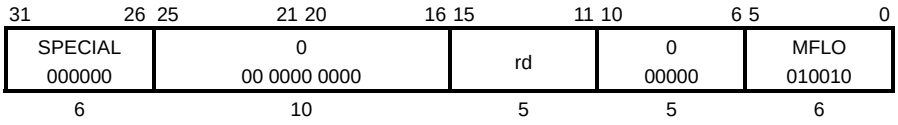
MFLO *rd*

Move From LO

Operation

rd ⇐ LO

Instruction Encoding



Description

The contents of the LO register is loaded into general-purpose register *rd*.

Exceptions

None

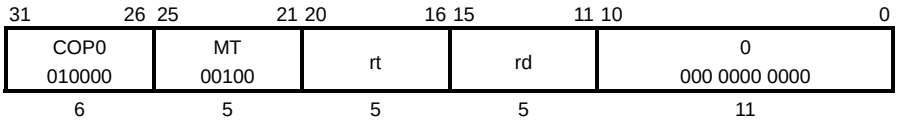
MTC0 *rt*, *rd*

Move To System Control Coprocessor (CP0)

Operation

Coprocessor register *rd* of CP0 ⇐ *rt*

Instruction Encoding



Description

The contents of general-purpose register *rt* is loaded into CP0 register *rd*.

The MTC0 instruction may not attempt to write to the Status register immediately before the RFE instruction. Otherwise, the contents of the Status register become undefined.

Likewise, the MTC0 instruction may not attempt to write to the Debug register immediately before the DERET instruction. Otherwise, the contents of the Debug register become 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

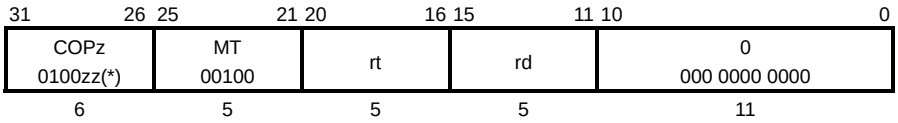
MTC*z* *rt*, *rd*

Move To Coprocessor z

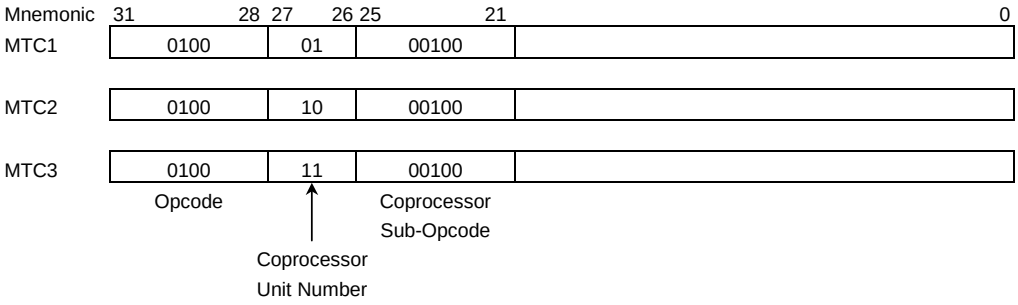
Operation

rt ⇐ coprocessor register *rd* of coprocessor unit *z*

Instruction Encoding



The following shows the opcode bit encoding. The two low-order bits in the opcode field signify the coprocessor unit number.



Description

The contents of general-purpose register *rt* is loaded into coprocessor register *rd* of coprocessor unit *z*.

Exceptions

Coprocessor Unusable exception

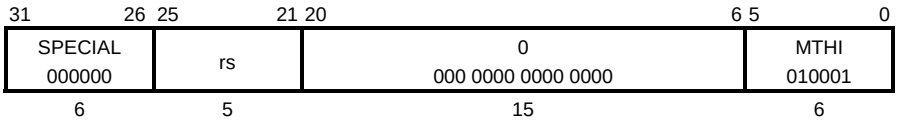
MTHI *rs*

Move To HI

Operation

HI ⇐ *rs*

Instruction Encoding



Description

The contents of general-purpose register *rs* is loaded into the HI register.

Exceptions

None

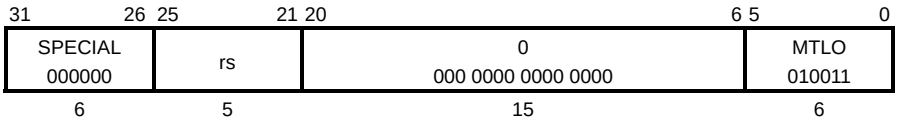
MTLO *rs*

Move To LO

Operation

LO ⇐ *rs*

Instruction Encoding



Description

The contents of general-purpose register *rs* is loaded into the LO register.

Exceptions

None

MULT (*rd*,) *rs*, *rt*

Multiply

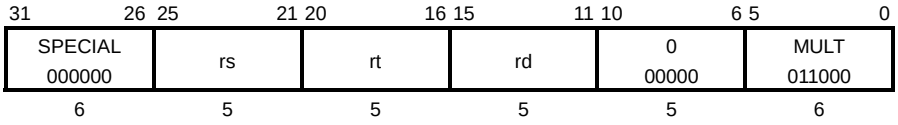
Operation

HI ⇐ high-order word of (*rs* × *rt*);

LO ⇐ low-order word of (*rs* × *rt*);

rd ⇐ low-order word of (*rs* × *rt*)

Instruction Encoding



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, causing the copy of the low-order word into a general-purpose register to be discarded.

No integer overflow exception occurs under any circumstances.

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

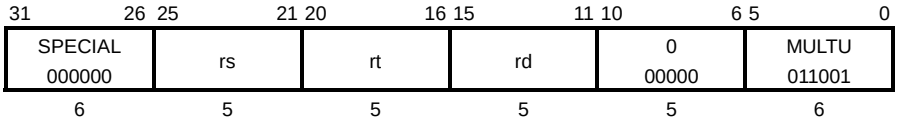
Operation

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, causing the copy of the low-order word into a general-purpose register to be discarded.

No integer overflow exception occurs under any circumstances.

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
```

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.

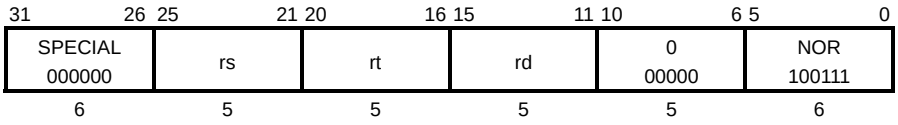
NOR *rd*, *rs*, *rt*

NOR

Operation

rd ⇐ *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

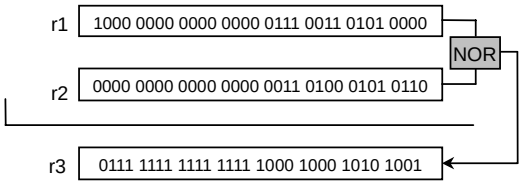
None

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.



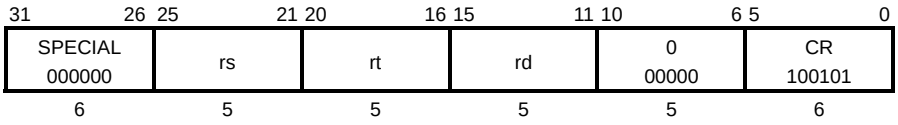
OR *rd*, *rs*, *rt*

OR

Operation

rd ⇐ *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

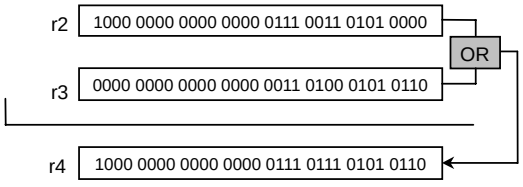
None

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.



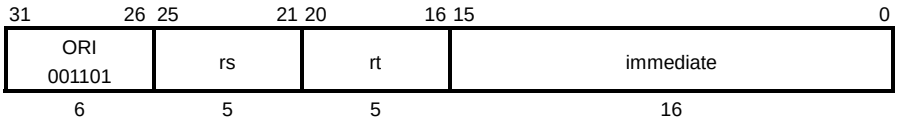
ORI *rt*, *rs*, *immediate*

OR Immediate

Operation

rt ⇐ *rs* OR *immediate*

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

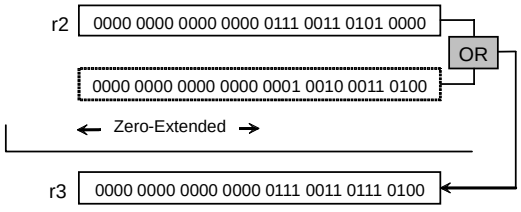
None

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.



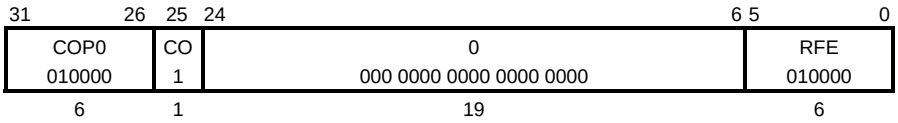
RFE

Restore From Exception

Operation

Status ⇐ Status[31:16] || Status[18:16] || Status[12:4] || Status[5:2]

Instruction Encoding



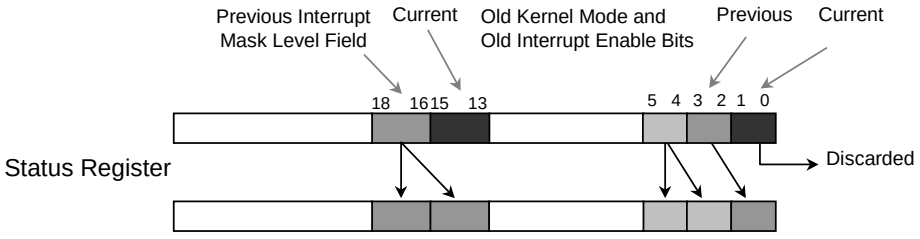
Description

RFE is an instruction for returning from an exception. The processor context in the Status register is restored to what it was before an exception was taken. The contents of the "old" Kernel Mode and Interrupt Enable bits (KUo/IEo) are popped to the "previous" bits (KUp/IEp), and the "previous" bits (KUp/IEp) are popped to the "current" bits (KUc/IEc). The "old" bits (KUo/IEo) remain unchanged. Additionally, the contents of the Previous Interrupt Mask Level field, PMask[2:0], is popped to the "current" field, CMask[2:0]. The PMask[2:0] field remains unchanged.

Typically, the RFE instruction is used in the jump delay slot of the JR instruction that restores the program counter (PC); it works elsewhere, however.

It is strictly prohibited to execute the RFE instruction immediately after an MTC0 instruction that writes to the Status register or immediately after an MFC0 instruction that reads from the Status register. Otherwise, the contents of the Status register become undefined.

The contents of the Status register become unpredictable if an interrupt occurs during execution of the RFE instruction. Therefore, all interrupts must be disabled prior to issuing the RFE instruction.



Exceptions

Coprocessor Unusable exception

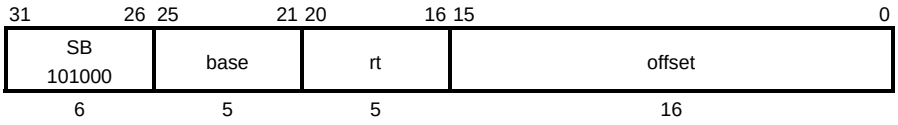
SB *rt*, *offset* (*base*)

Store Byte

Operation

$rt \Rightarrow \{offset\ (base)\}$

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

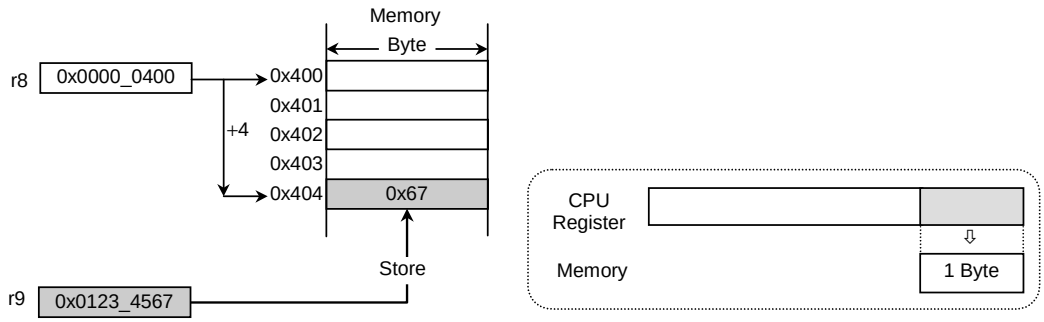
Address Error exception

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.



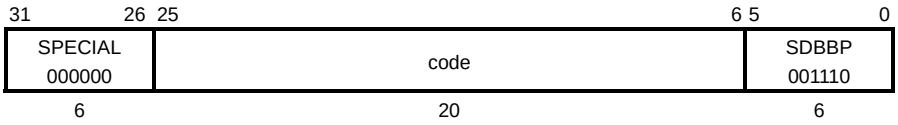
SDBBP *code*

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 while a Debug exception is being serviced (i.e., the DM bit in the Debug register is set). The operation of the SDBBP instruction is undefined when DM=1.

The SDBBP instruction may not be used within the user's program; it is intended for use by development systems.

Exceptions

Debug Breakpoint exception

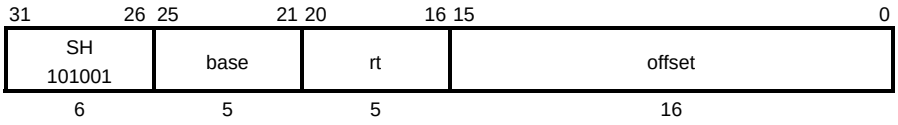
SH *rt*, *offset* (*base*)

Store Halfword

Operation

$rt \Rightarrow \{offset\ (base)\}$

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:

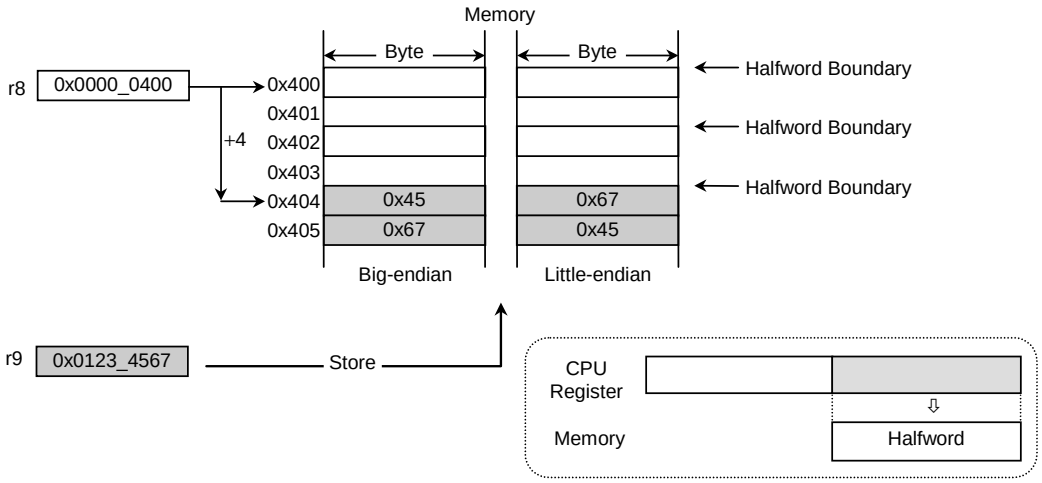
```
SH r9,4(r8)
```

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
0x400
0x401
0x402
0x403
0x404
0x405
+4
0x45
0x67
0x67
0x45
Halfword Boundary
Halfword Boundary
Halfword Boundary
Big-endian
Little-endian
r9
0x0123_4567
Store
CPU
Register
Memory
Halfword

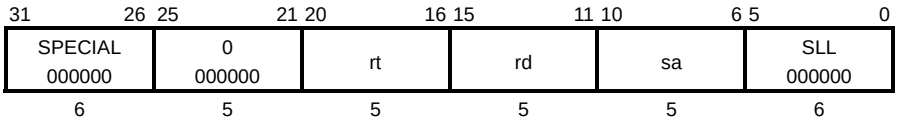
SLL *rd*, *rt*, *sa*

Shift Left Logical

Operation

$rd \Leftarrow rt << sa$

Instruction Encoding



Description

The 32-bit 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

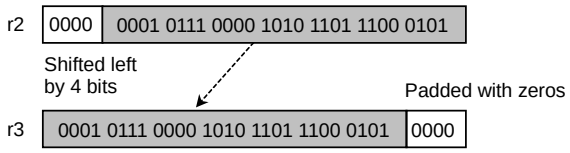
None

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.



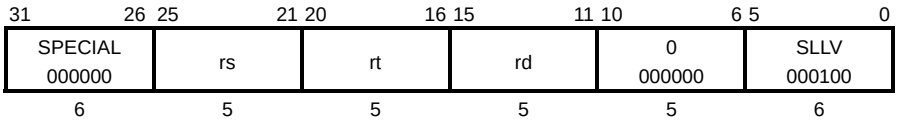
SLLV *rd*, *rt*, *rs*

Shift Left Logical Variable

Operation

rd ⇐ *rt* << 5 LSBs of *rs*

Instruction Encoding



Description

The 32-bit 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

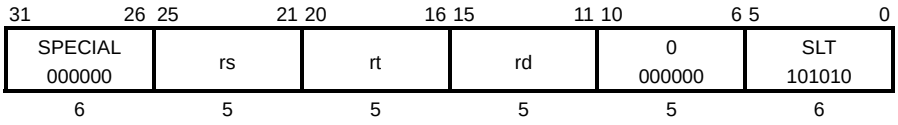
SLT *rd*, *rs*, *rt*

Set On Less Than

Operation

if *rs* < *rt* then *rd* ⇐ 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 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

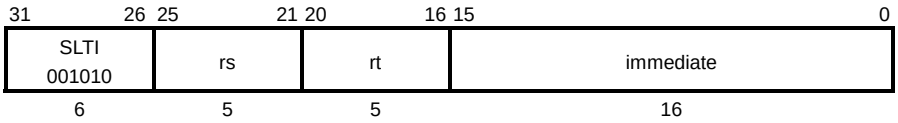
SLTI *rt*, *rs*, *immediate*

Set On Less Than Immediate

Operation

if *rs* < *immediate* then *rt* ⇐ 1; else *rt* ⇐ 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 signed integers. If *rs* is less than the *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. This gives a range of -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

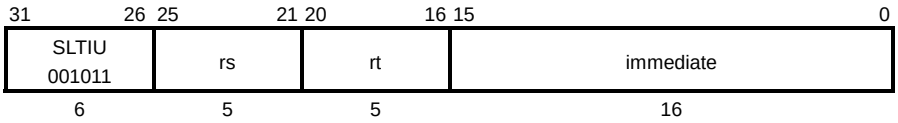
SLTIU *rt*, *rs*, *immediate*

Set On Less Than Immediate Unsigned

Operation

if *rs* < *immediate* then *rt* ⇐ 1; else *rt* ⇐ 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 the *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

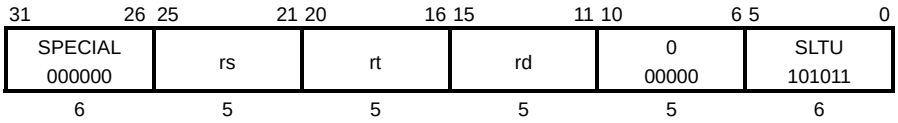
SLTU *rd*, *rs*, *rt*

Set On Less Than Unsigned

Operation

if *rs* < *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

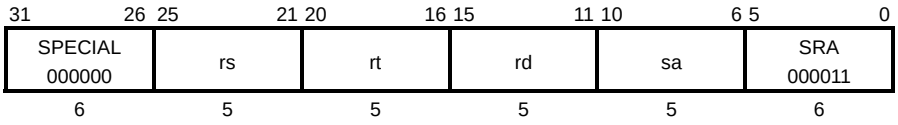
SRA *rd*, *rt*, *sa*

Shift Right Arithmetic

Operand

rd ⇐ *rt* >> *sa*

Instruction Encoding



Description

The 32-bit 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

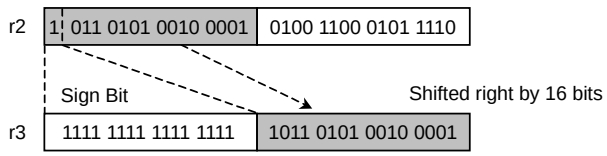
None

Example

Assume that register r2 contains 0xB521_4C5E. Then, executing the instruction:

```
SRA r3,r2,16
```

places 0xFFFF_B521 into r3, as shown below.



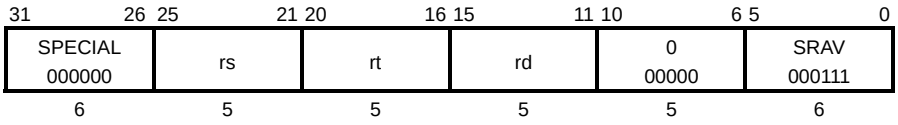
SRAV *rd*, *rt*, *rs*

Shift Right Arithmetic Variable

Operation

rd ⇐ *rt* >> 5 LSBs of *rs*

Instruction Encoding



Description

The 32-bit 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

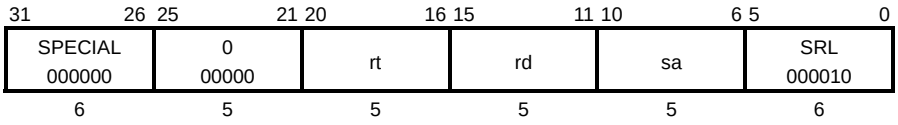
SRL *rd*, *rt*, *sa*

Shift Right Logical

Operation

rd ⇐ *rt* >> *sa*

Instruction Encoding



Description

The 32-bit 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

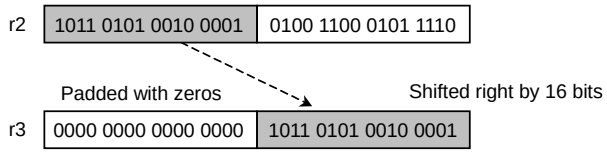
None

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.



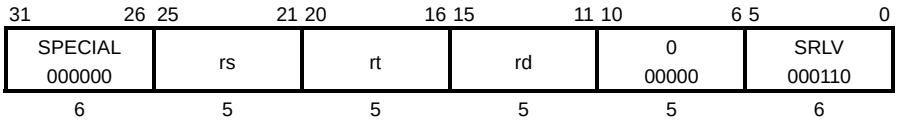
SRLV *rd*, *rt*, *rs*

Shift Right Logical Variable

Operation

rd ⇐ *rt* >> 5 LSBs of *rs*

Instruction Encoding



Description

The 32-bit 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*. Zeros are supplied to the vacated positions on the left. The result is placed into general-purpose register *rd*.

Exceptions

None

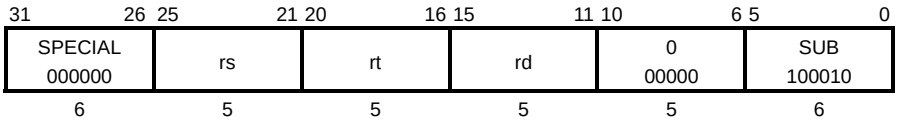
SUB *rd*, *rs*, *rt*

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

Examples

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's-complement 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.

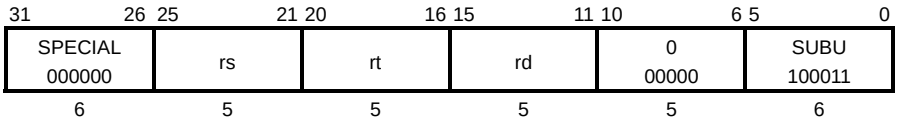
SUBU *rd*, *rs*, *rt*

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

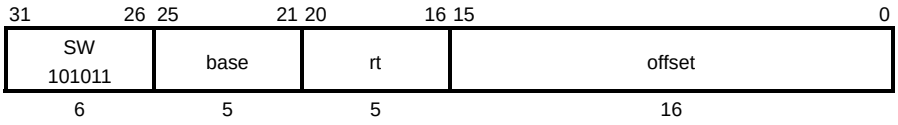
SW *rt*, *offset* (*base*)

Store Word

Operation

$rt \Rightarrow \{offset\ (base)\}$

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 are 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:

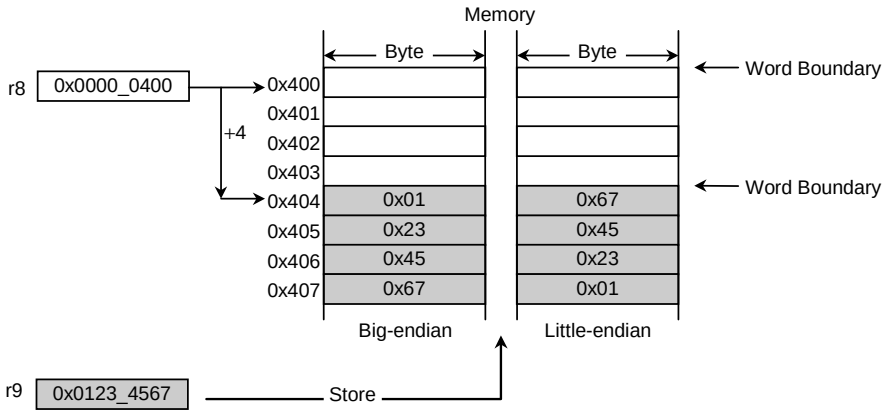
```
SW r9,4(r8)
```

stores 0x12, 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 halfword boundary.


Memory
Byte
Byte
Word Boundary
r8
0x0000_0400
+4
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
0x67
0x45
0x23
0x01
Word Boundary
Big-endian
Little-endian
r9
0x0123_4567
Store

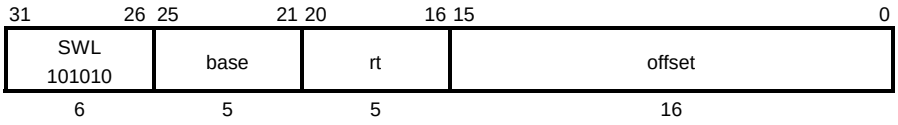
SWL *rt*, *offset* (*base*)

Store Word Left

Operation

$rt \Rightarrow \{offset\ (base)\}$

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.

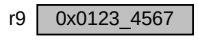
The SWL and SWR instructions are used in combination to store a word into memory locations that are not on a natural word boundary.

Exceptions

Address Error exception

Example

Assume that registers r8 and r9 contain 0x0000_0400 and 00123_4567 respectively.

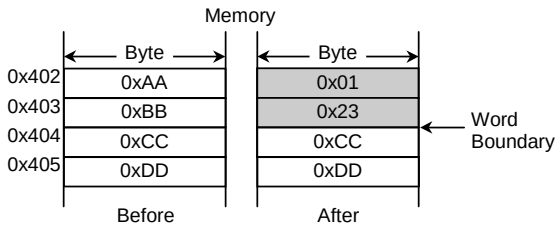


- Big-endian mode

  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.



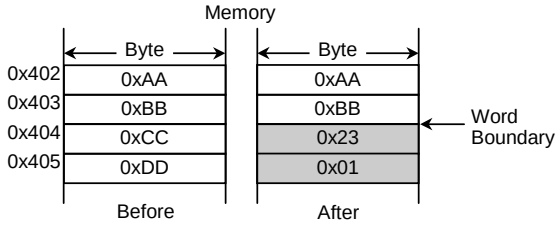
(a) Big-endian

- 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

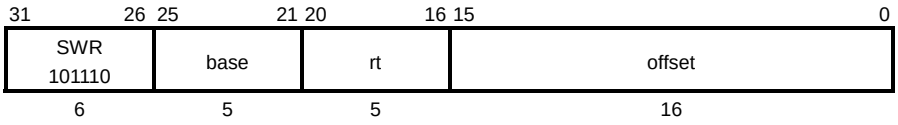
SWR *rt*, *offset* (*base*)

Store Word Right

Operation

$rt \Rightarrow \{offset\ (base)\}$

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.

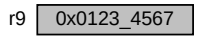
The SWL and SWR instructions are used in combination to store a word into memory locations that are not on a natural word boundary.

Exceptions

Address Error exception

Example

Assume register r9 contains 0x123_4567.



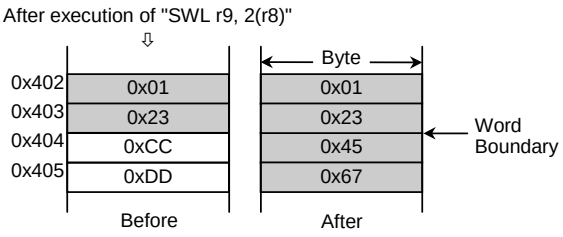
The following shows how to store the right portion of a general-purpose register after storing the left portion as described on the previous SWL pages.

- 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

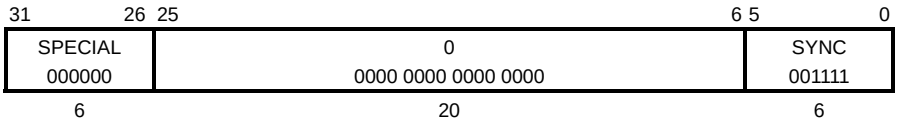
SYNC

Synchronize

Operation

Synchornize operation

Instruction Encoding



Description

The SYNC instruction interlocks the instruction pipeline until loads and stores performed prior to the present instruction are completed before loads or stores 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 TX19 continues to execute subsequent instructions. This is called *nonblocking loads*. All the other parts of the pipeline can continue to work on non-dependent instructions.

The SYNC instruction is allowed in User mode.

Exceptions

None

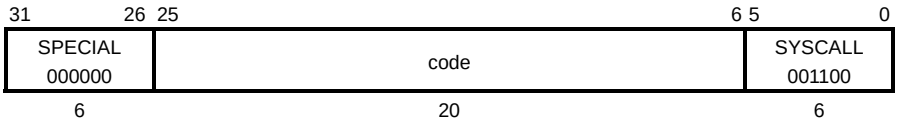
SYSCALL *code*

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

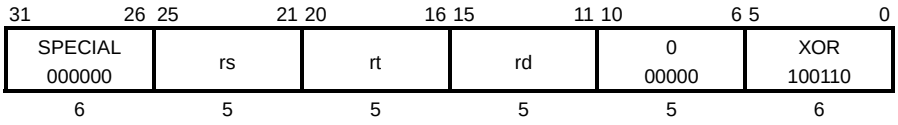
XOR *rd*, *rs*, *rt*

Exclusive OR

Operation

rd ⇐ *rs* 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 into general-purpose register *rd*.

Exceptions

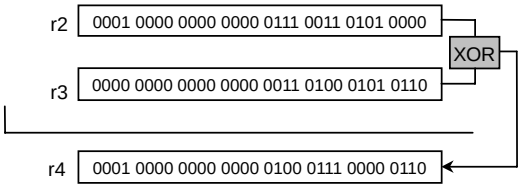
None

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 in r4, as shown below.



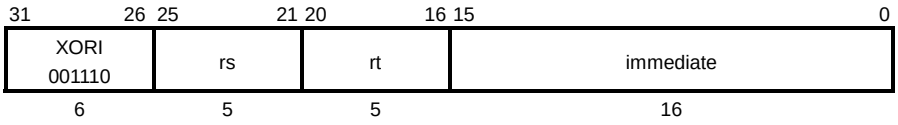
XORI *rt*, *rs*, *immediate*

Exclusive OR Immediate

Operation

rt ⇐ *rs* XOR *immediate*

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 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 XOR instruction (see Section 3.3.2, *32-Bit Constants*).

Exceptions

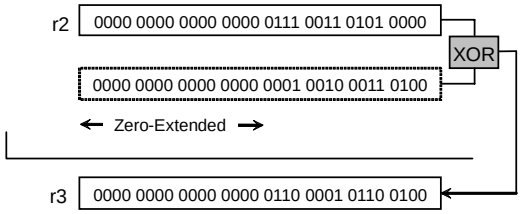
None

Example

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

```
XORI r3,r2,0x1234
```

places 0x0000_6164 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*.

All the instructions in the 16-bit ISA consist of 16 bits with the exception of JAL and JALX which are 32-bits wide. Generally, each 16-bit instruction corresponds to exactly one 32-bit instruction. The 16-bit instructions fetched from the memory subsystem are translated to 32-bit instructions on the fly by relatively simple translation hardware called MIPS16 decompressor. This is done serially as a preprocessor before the standard instruction decoder. Remember that there are a few 16-bit instructions whose functions are slightly different from the 32-bit equivalents. Each instruction page in this appendix shows both the instruction codes before and after decompression.

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, certain instructions can use r24 (t8), r29 (sp) and r31 (ra). r24 serves as the condition code register for handling compare results. r29 maintains the program stack pointer. r31 is the link register to store the subroutine return address. These registers are implicitly referred to through special function codes.

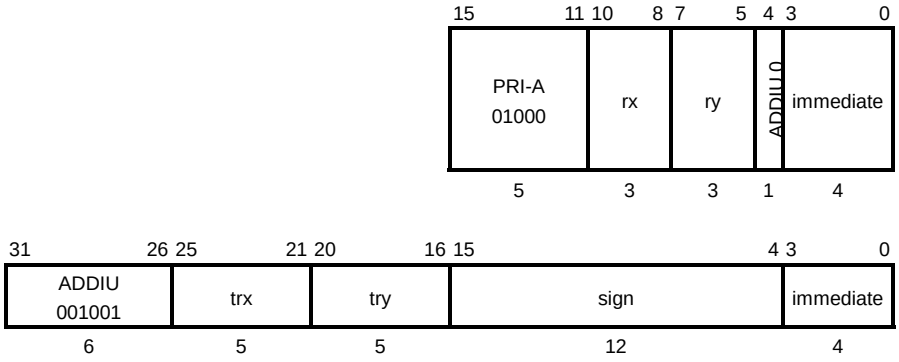
ADDIU *ry*, *rx*, *immediate*

Add Immediate Unsigned

Operation

ry ⇐ *rx* + *immediate*

Instruction Encoding

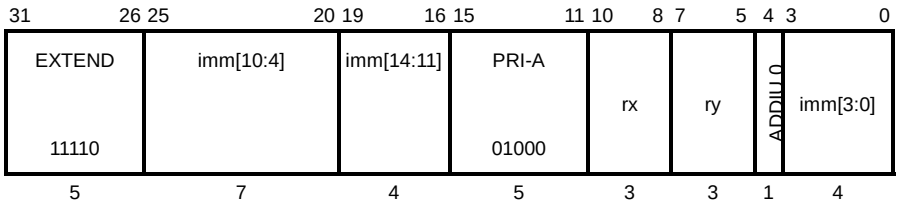


Description

Although the opcode stands for "Add Immediate Unsigned," 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.

The *immediate* field is 4 bits in length. This gives a range of -8 to +7. If the *immediate* is outside this range, the instruction is EXTENDed. EXTEND extends the immediate field in the ALU immediate instructions to 16 bits, with the exception of this instruction. ADDIU has a 4-bit immediate field, but since EXTEND can only supply 11 more bits, the wider immediate is limited to 15 bits. Thus, the EXTENDEed immediate field allows a 15-bit signed immediate in the range of -16384 to +16833. The EXTENDed instruction code is given below.



Exceptions

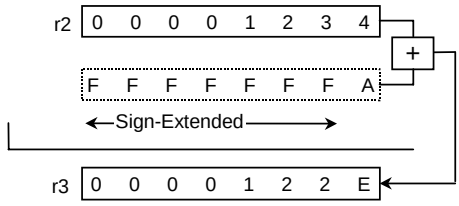
None

Example

Assume that register r2 contains 0x0000_1234. Then, executing the instruction:

```
ADDIU r3,r2,-6
```

places the sum 0x0000_122E into r3.



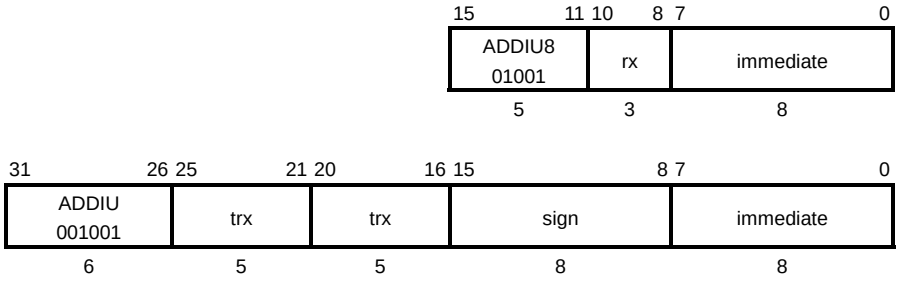
ADDIU *rx, immediate*

Add Immediate Unsigned

Operation

rx ⇐ *rx* + *immediate*

Instruction Encoding



Description

Although the opcode stands for "Add Immediate Unsigned," 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.

The *immediate* field is 8 bits in length. This gives a range of -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

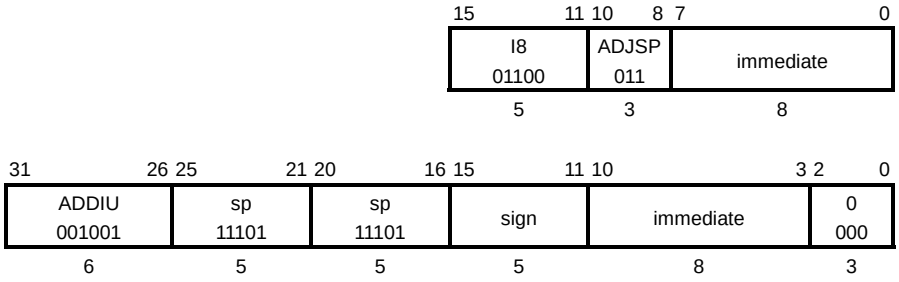
ADDIU sp, *immediate*

Add Immediate Unsigned

Operation

sp ⇐ sp + *immediate*

Instruction Encoding



Description

Although the opcode stands for "Add Immediate Unsigned," 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.

The *immediate* field is 8 bits in length. Shifted three bits, this gives a range of -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

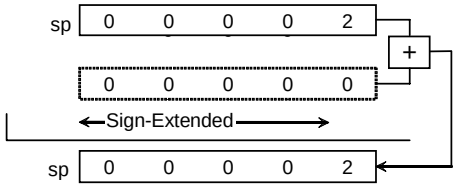
None

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.


sp
0 0 0 0 2
+
0 0 0 0 0
Sign-Extended
sp
0 0 0 0 2

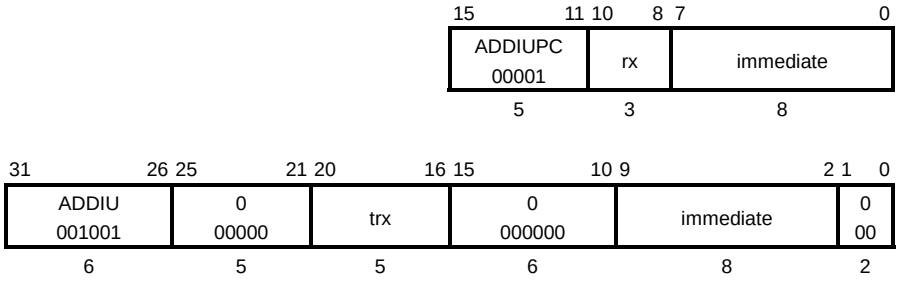
ADDIU *rx*, pc, *immediate*

Add Immediate Unsigned

Operation

rx ⇐ Masked base PC + *immediate*

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 base PC value 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.

Zeros fill in bits 25 to 21 as placeholders. 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.

The *immediate* field is 8 bits in length. Shifted two bits, this gives a range of 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 or 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

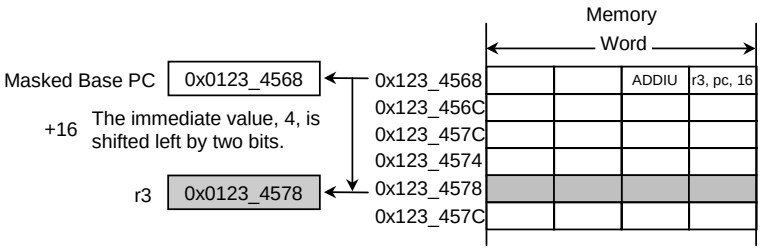
Exceptions

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 MIPS16 decompressor, 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.



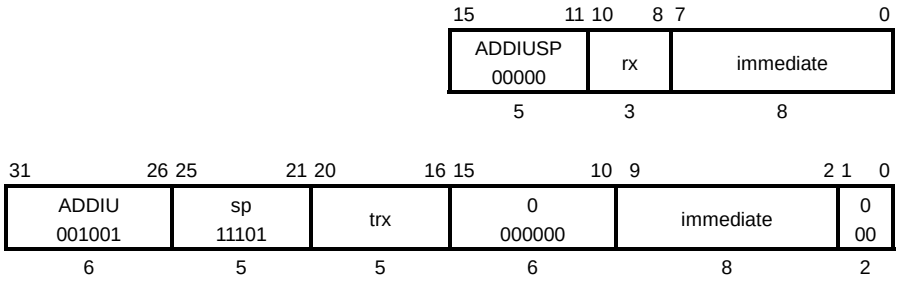
ADDIU *rx*, sp, *immediate*

Add Immediate Unsigned

Operation

rx ⇐ sp + *immediate*

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.

The *immediate* field is 8 bits in length. Shifted two bits, this gives a range of 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

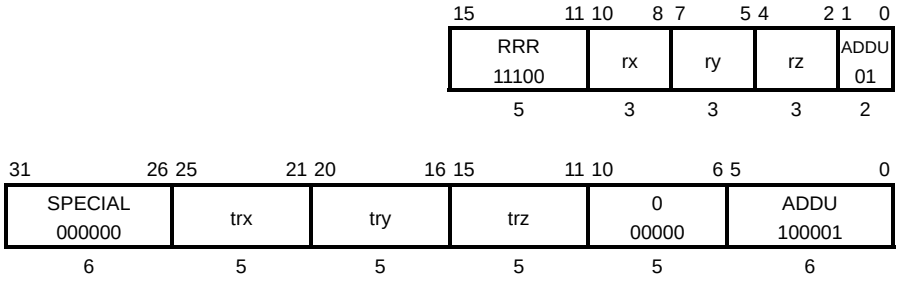
ADDU *rz*, *rx*, *ry*

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 0x0200_0000 and 0x0123_4567 respectively. Then, executing the instruction:

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

places the sum (0x0323_4567) into r4.

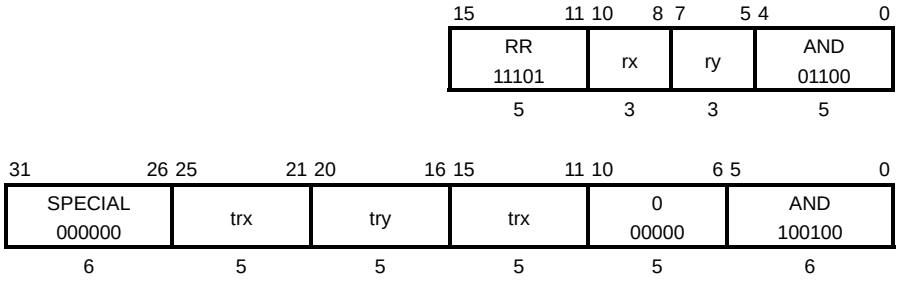
AND *rx*, *ry*

AND

Operation

rx ⇐ *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

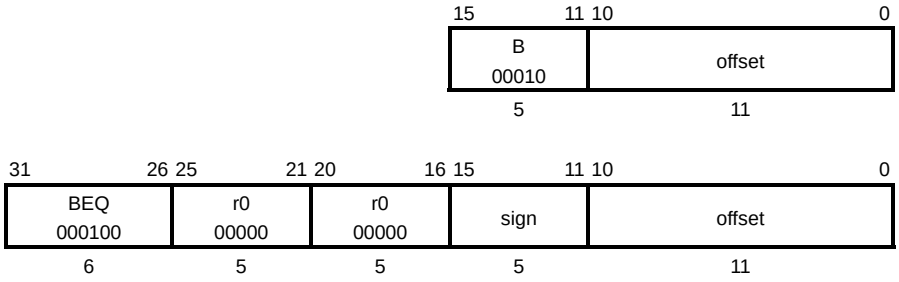
B *offset*

Branch Unconditional

Operation

pc ⇐ pc + *offset*

Instruction Encoding



Description

The program unconditionally branches to the target address with a delay of one instruction (i.e., two instruction cycles). See Section 5.3.4, *Branch Instructions (16-Bit ISA)*, for pipeline delays. The target address is computed relative to the address of the immediately following instruction (PC+2); the 11-bit immediate *offset* is shifted left by one bit, sign-extended and added to PC+2 to form the target address.

This instruction is implemented as a 32-bit BEQ instruction that compares r0 and r0, causing an unconditional branch. However, the operation of this instruction differs from that of the 32-bit BEQ instruction in that the B instruction does not have a delay slot; i.e., the branch always takes effect before the next instruction.

The *offset* field is 11 bits in length. This gives a range of -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 adress is computed in the same manner.

Exceptions

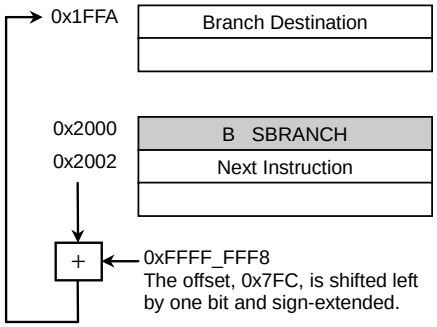
None

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 offset operand 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.



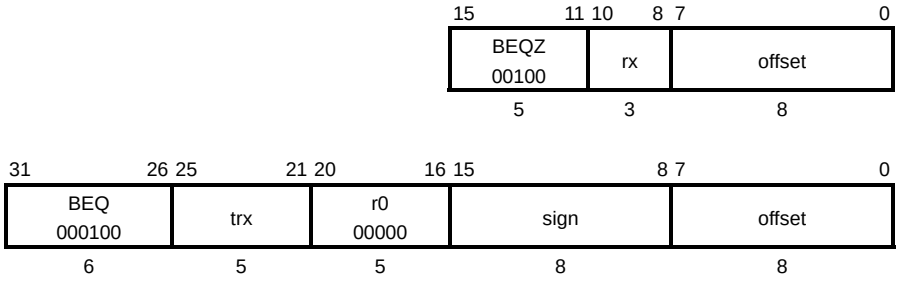
BEQZ *rx*, *offset*

Branch On Equal To Zero

Operation

if *rx* = 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction cycles). See Section 5.3.4, *Branch Instructions (16-Bit ISA)*, for pipeline delays. The target address is computed relative to the address of the immediately following instruction (PC+2); the 8-bit immediate *offset* is shifted left by one bit, sign-extended and added to PC+2 to form the target address.

The operation of this instruction differs from that of the corresponding 32-bit BEQ instruction in that the 16-bit BEQZ instruction does not have a delay slot.

The *offset* field is 8 bits in length. This gives a range of -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

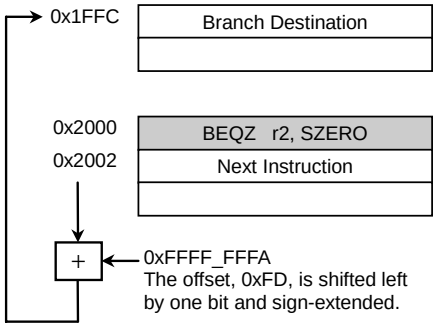
Example

```
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 offset operand 0xFD (see the figure below). Thus the instruction code for this branch instruction becomes 0x22FD.

If the contents of r2 is equal to zero, the processor transfers program control to address 0x1FFC.

Otherwise, the program just continues to the next instruction at 0x2002.



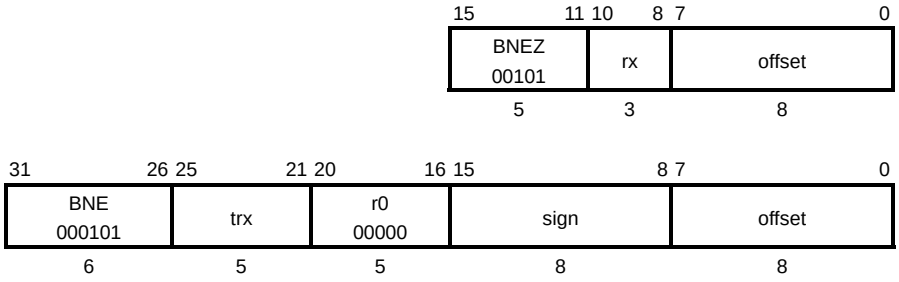
BNEZ *rx*, *offset*

Branch On Not Equal To Zero

Operation

if $rx \neq 0$ then pc $\Leftarrow$ pc + *offset*

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 (i.e., two instruction cycles). See Section 5.3.4, *Branch Instructions (16-Bit ISA)*, for pipeline delays. The target address is computed relative to the address of the immediately following instruction (PC+2); the 8-bit immediate *offset* is shifted left by one bit, sign-extended and added to PC+2 to form the target address.

The operation of this instruction differs from that of the corresponding 32-bit BNE instruction in that the 16-bit BNEZ instruction does not have a delay slot.

The *offset* field is 8 bits in length. This gives a range of -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

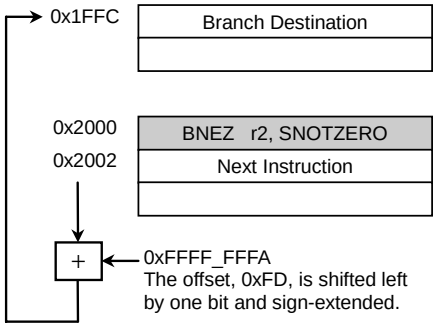
Example

```
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 offset operand 0xFD (see the figure below). Thus the instruction code for this branch instruction becomes 0x2AFD.

If the contents of r2 is not equal to zero, the processor transfers program control to address 0x1FFC.

Otherwise, the program just continues to the next instruction at 0x2002.



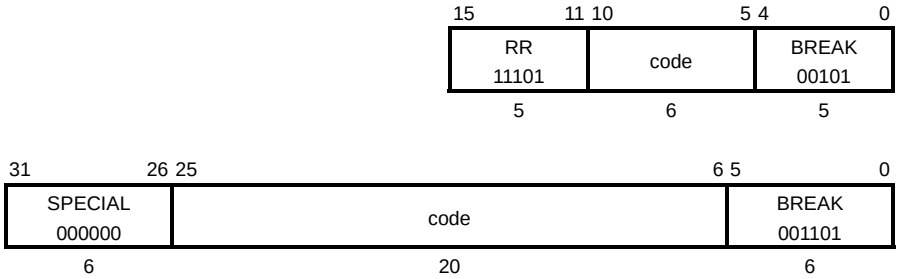
BREAK *code*

Breakpoint Exception

Operation

Breakpoint exception

Instruction Encoding



Description

When this instruction is executed, a breakpoint trap 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

None

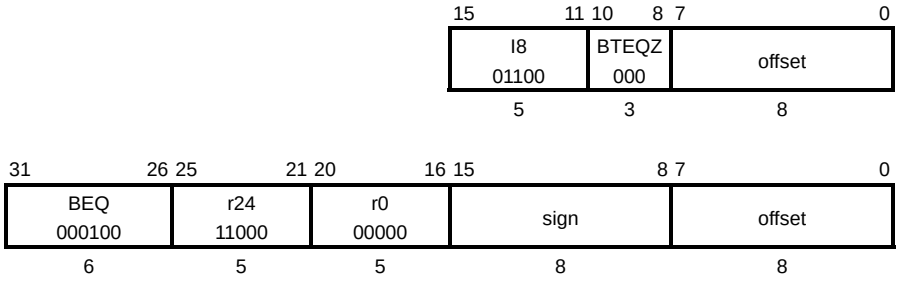
BTEQZ *offset*

Branch On T8 Equal To Zero

Operation

if t8 = 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction cycles). See Section 5.3.4, *Branch Instructions (16-Bit ISA)*, for pipeline delays. The target address is computed relative to the address of the immediately following instruction (PC+2); the 8-bit immediate *offset* is shifted left by one bit, sign-extended and added to PC+2 to form the target address.

The operation of this instruction differs from that of the corresponding 32-bit BEQ instruction in that the 16-bit BTEQZ instruction does not have a delay slot.

The *immediate* field is 8 bits in length. This gives a range of -256 to +254. If the *immediate* 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

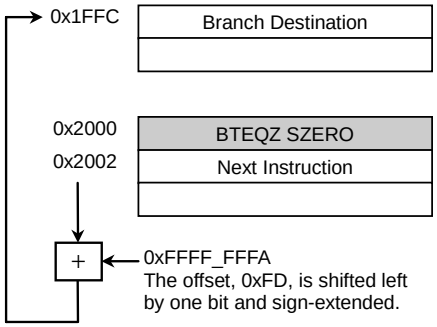
None

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 offset operand 0xFD (see the figure below). Thus the instruction code for this branch instruction becomes 0x60FD.

If the contents of t8 is equal to zero, the processor transfers program control to address 0x1FFC. Otherwise, the program just continues to the next instruction at 0x2002.



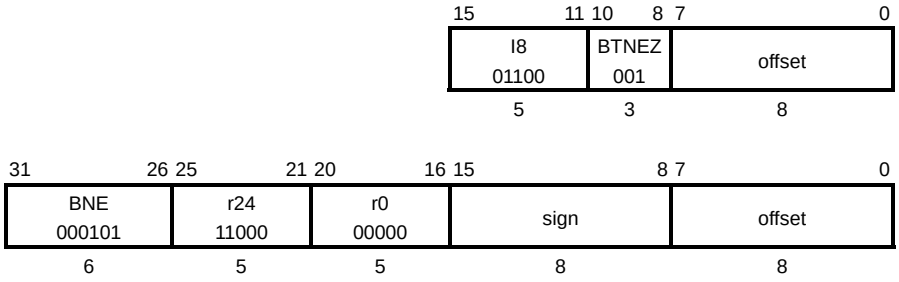
BTNEZ *offset*

Branch On T8 Not Equal To Zero

Operation

if t8 ≠ 0 then pc ⇐ pc + *offset*

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 (i.e., two instruction cycles). See Section 5.3.4, *Branch Instructions (16-Bit ISA)*, for pipeline delays. The target address is computed relative to the address of the immediately following instruction (PC+2); the 8-bit immediate *offset* is shifted left by one bit, sign-extended and added to PC+2 to form the target address.

The operation of this instruction differs from that of the corresponding 32-bit BNE instruction in that the 16-bit BTNEZ instruction does not have a delay slot.

The *immediate* field is 8 bits in length. This gives a range of -256 to +254. If the *immediate* 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 adess is computed in the same manner.

Exceptions

None

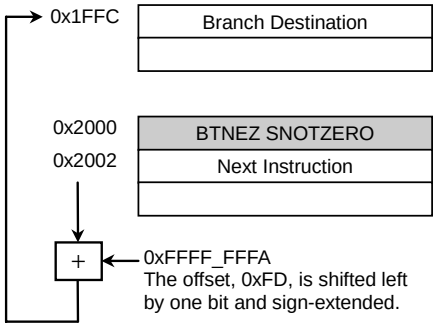
Example

```
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 offset operand 0xFD (see the figure below). Thus the instruction code for this branch instruction becomes 0x61FD.

If the contents of t8 is equal to zero, the processor transfer program control to address 0x1FFC.

Otherwise, the program just continues to the next instruction at 0x2002.



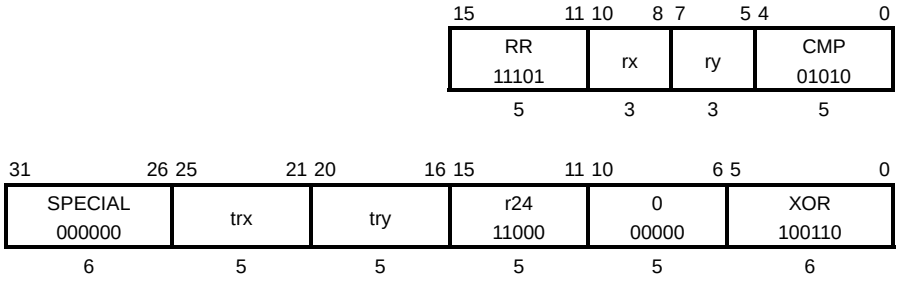
CMP *rx*, *ry*

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

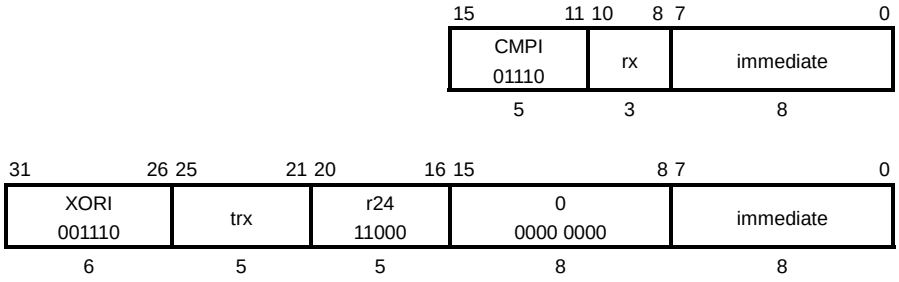
CMPI *rx*, *immediate*

Compare Immediate

Operation

if *rx* = *immediate* 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.

The *immediate* field is 8 bits in length. This gives a range of 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

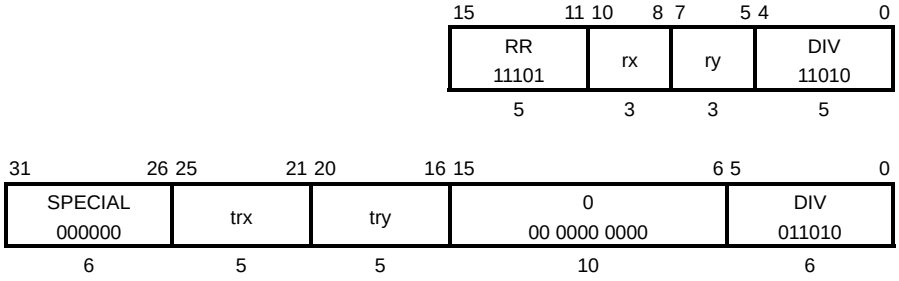
DIV *rx*, *ry*

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 integer overflow exceptions.

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 cache misses, delay cycles and exceptions occur.

If the divide instruction is followed by an MFHI, MFLO, MADD or MADDU 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

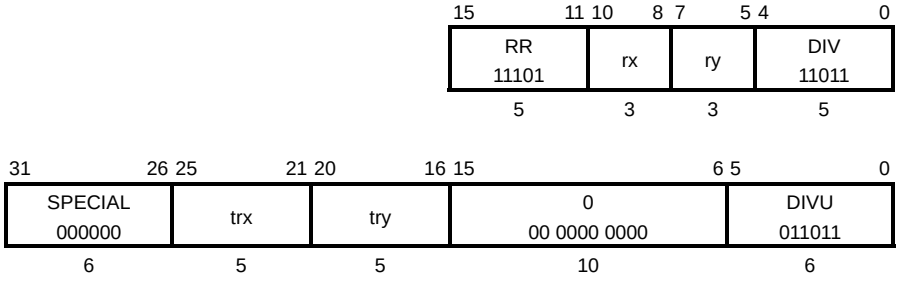
DIVU *rx*, *ry*

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 DIVU 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

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, causing the processor to perform the fetch in two steps. The program unconditionally jumps to the target address with a delay of one instruction (i.e., two instruction 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+2). The 26-bit *target* is shifted left by two bits and combined with the four most-significant bits of PC+2 to form the target address. The JAL instruction never toggles the ISA mode bit of the program counter (PC).

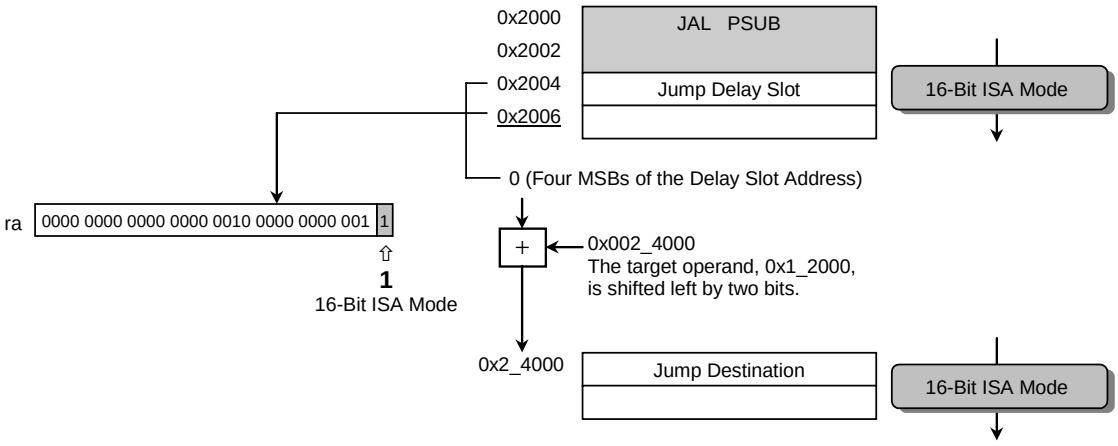
The address of the instruction after the jump delay slot is saved in the link register, ra (r31). The ISA mode specifier (i.e., a 1 for the 16-bit ISA mode) is saved in the least-significant bit of ra.

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 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 ra together with the ISA mode bit value; thus the ra value becomes 0x0000_2007.


0x2000
0x2002
0x2004
0x2006
JAL PSUB
Jump Delay Slot
16-Bit ISA Mode
0 (Four MSBs of the Delay Slot Address)
ra
0000 0000 0000 0000 0010 0000 0000 001
1
1
16-Bit ISA Mode
+
0x002_4000
The target operand, 0x1_2000,
is shifted left by two bits.
0x2_4000
Jump Destination
16-Bit ISA Mode

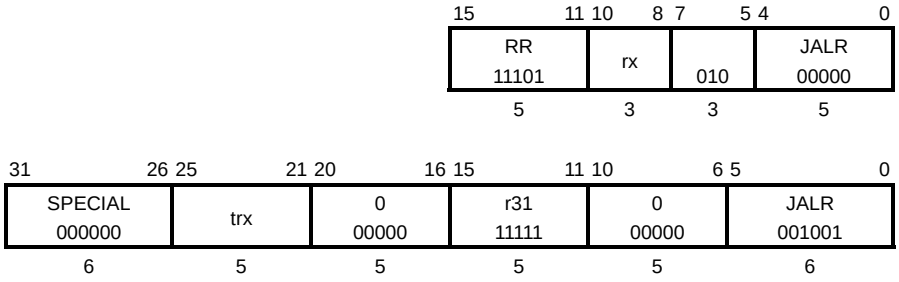
JALR ra, *rx*

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 (i.e., two instruction 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), together 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 these low-order bits are not zero, an Address Error exception will occur when the processor fetches the instruction at the jump destination.

Exceptions

None

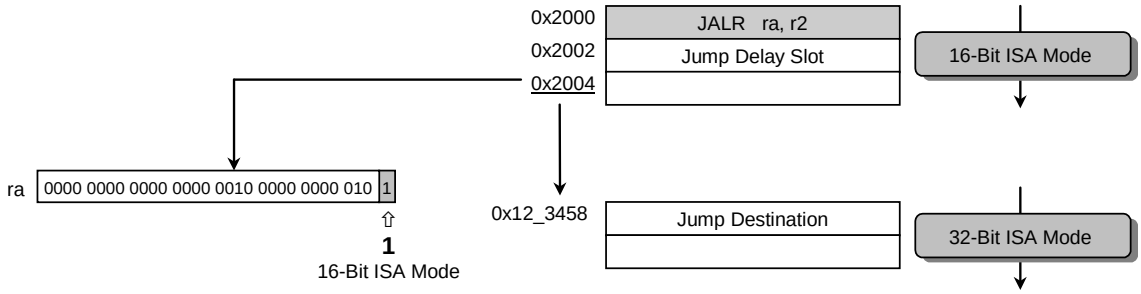
Example

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. The jump takes effect after the instruction in the jump delay slot is executed. 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 together with the ISA mode bit value; thus the ra value becomes 0x0000_2005.



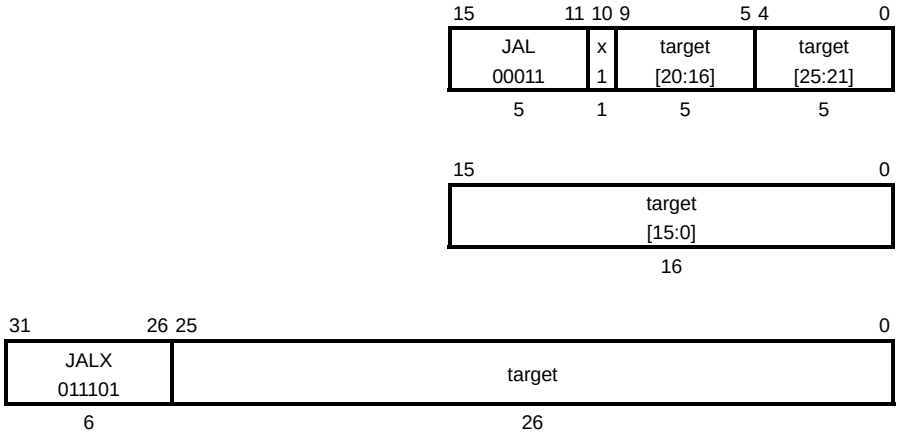
JALX *target*

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, causing the processor to perform the fetch in two steps. The program unconditionally jumps to the target address with a delay of one instruction (i.e., two instruction 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+2). The 26-bit target is shifted left by two bits and combined with the four most-significant bits of PC+2 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

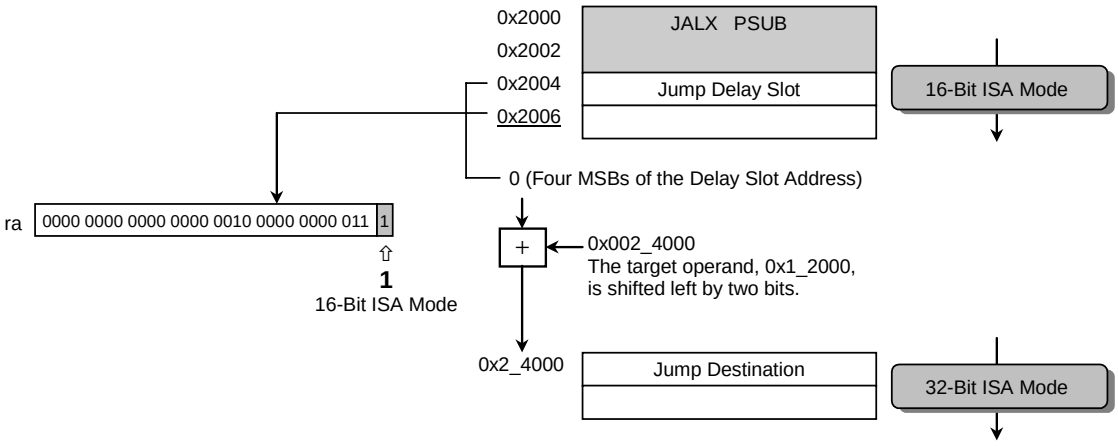
None

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 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 32-bit ISA mode. The address of the instruction after the jump delay slot is saved in ra together with the ISA mode bit value; thus the ra value becomes 0x0000_2007.



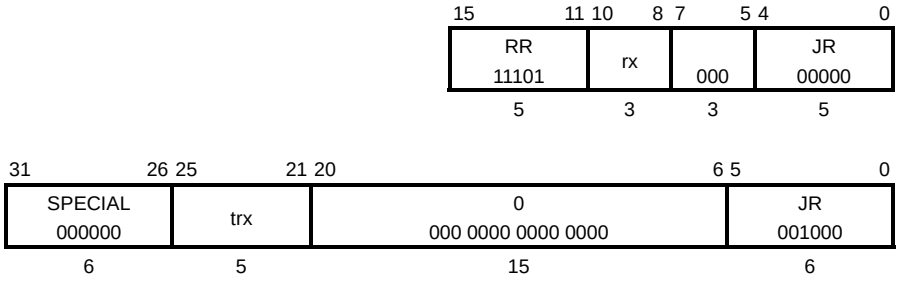
JR *rx*

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 (i.e., two instruction 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 these low-order bits are not zero, an Address Error exception will occur when the processor fetches the instruction at the jump destination.

Exceptions

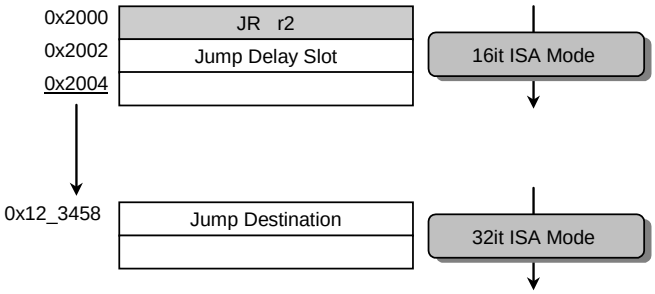
None

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.


0x2000
0x2002
0x2004
JR r2
Jump Delay Slot
16it ISA Mode
0x12_3458
Jump Destination
32it ISA Mode

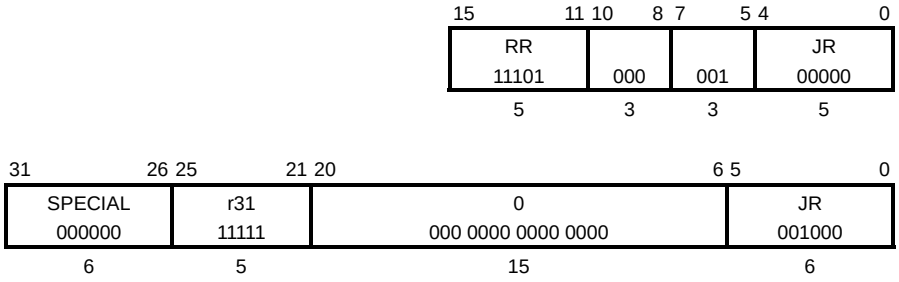
JR ra

Jump Register

Operation

pc ⇐ ra

Instruction Encoding



Description

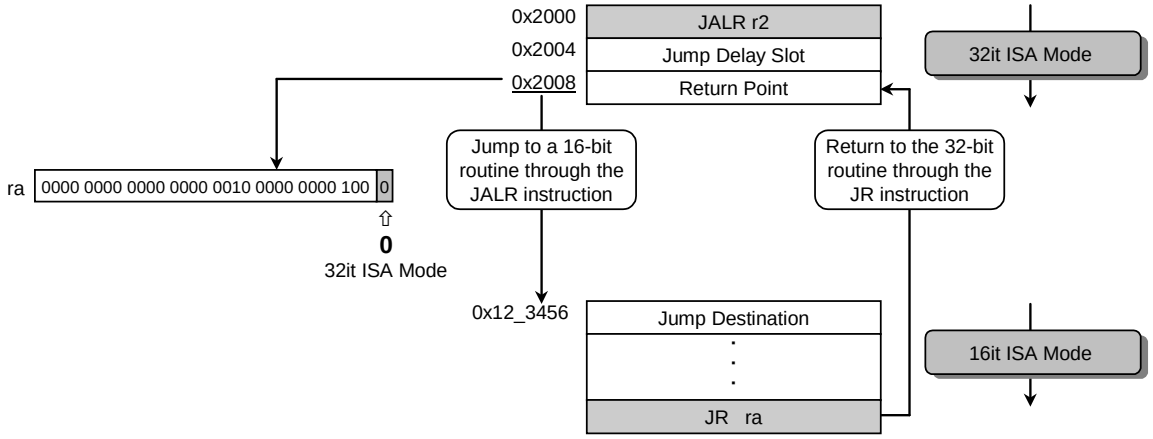
The program unconditionally jumps to the address contained in the link register, ra (r31), with the least-significant bit cleared, with a delay of one instruction (i.e., instruction cycles). The least-significant bit of ra is interpreted as the ISA mode specifier.

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.


0x2000
0x2004
0x2008
JALR r2
Jump Delay Slot
Return Point
32it ISA Mode
Jump to a 16-bit routine through the JALR instruction
Return to the 32-bit routine through the JR instruction
ra
0000 0000 0000 0000 0010 0000 0000 100
0
0
32it ISA Mode
0x12_3456
Jump Destination
JR ra
16it ISA Mode

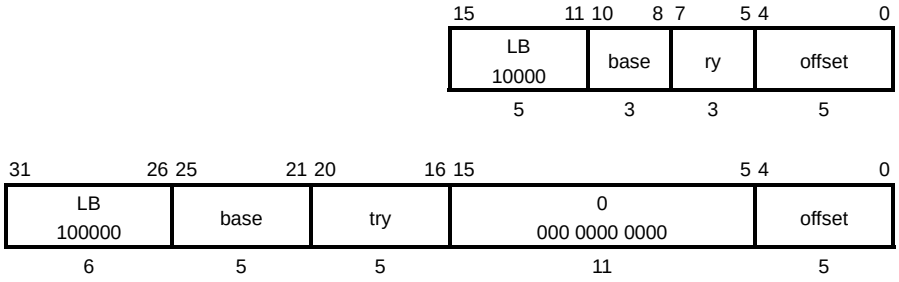
LB *ry*, *offset* (*base*)

Load Byte

Operation

ry ⇐ {*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 EA is sign-extended and loaded into general-purpose register *ry*.

The *offset* field is 5 bits in length. This gives a range of 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

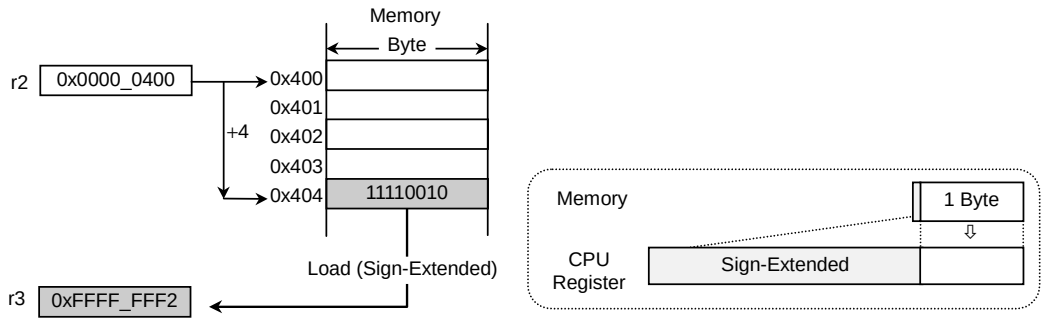
Address Error exception

Example

Assume that register r2 contains 0x_0000_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.


Memory
Byte
r2
0x0000_0400
0x400
0x401
0x402
0x403
0x404
+4
11110010
Load (Sign-Extended)
r3
0xFFFF_FFF2
Memory
1 Byte
CPU
Register
Sign-Extended

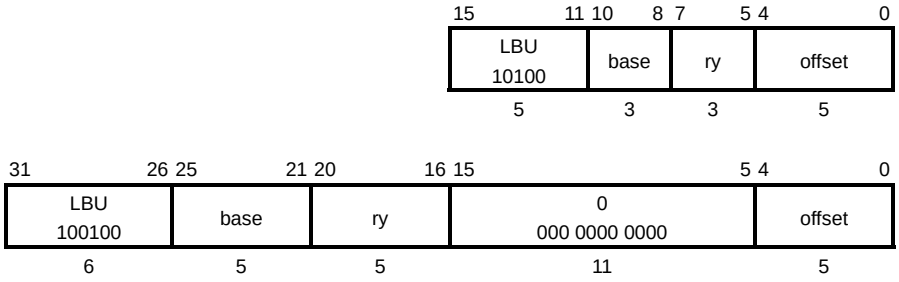
LBU *ry*, *offset* (*base*)

Load Byte Unsigned

Operation

ry ⇐ {*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 EA is zero-extended and loaded into general-purpose register *ry*.

The *offset* field is 5 bits in length. This gives a range of 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

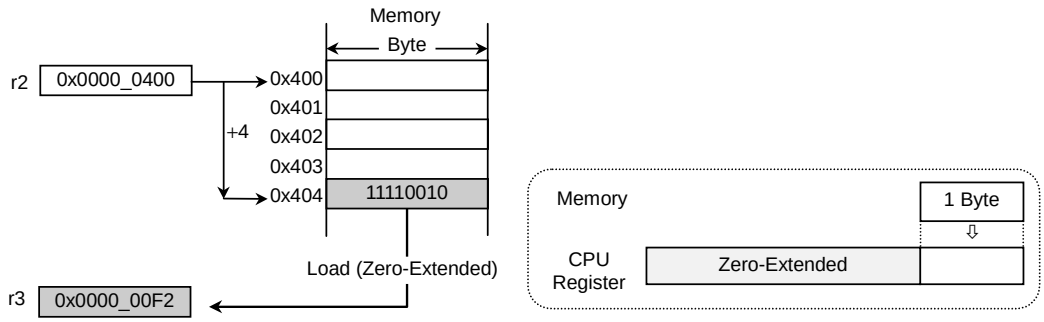
Address Error exception

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.


Memory
Byte
r2
0x0000_0400
0x400
0x401
0x402
0x403
0x404
+4
11110010
Load (Zero-Extended)
r3
0x0000_00F2
Memory
1 Byte
CPU
Register
Zero-Extended

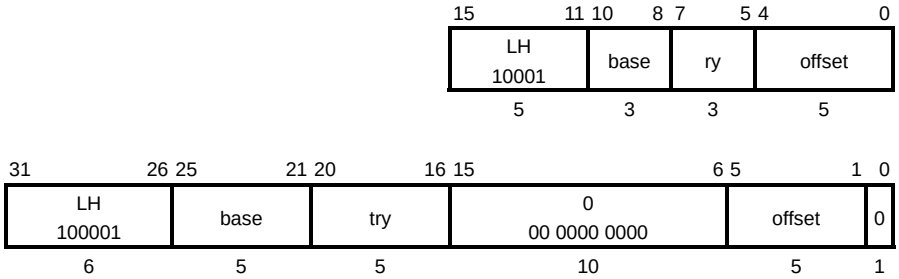
LH *ry*, *offset* (*base*)

Load Halfword

Operation

ry ⇐ {*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 EA is sign-extended and loaded into general-purpose register *ry*.

The *offset* field is 5 bits in length. Shifted one bit, this gives a range of 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

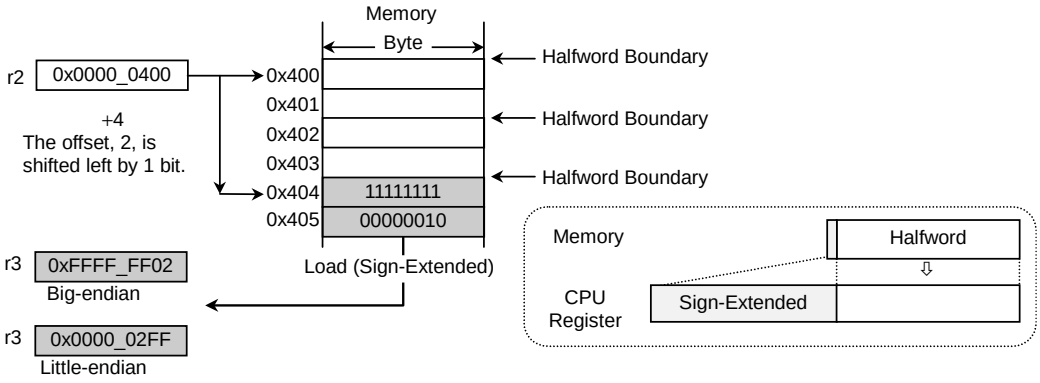
Address Error exception

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 MIPS16 decompressor, 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.


Memory
Byte
r2
0x0000_0400
0x400
0x401
0x402
0x403
0x404
0x405
11111111
00000010
Halfword Boundary
Halfword Boundary
Halfword Boundary
+4
The offset, 2, is
shifted left by 1 bit.
r3
0xFFFF_FF02
Big-endian
r3
0x0000_02FF
Little-endian
Load (Sign-Extended)
Memory
Halfword
CPU
Register
Sign-Extended

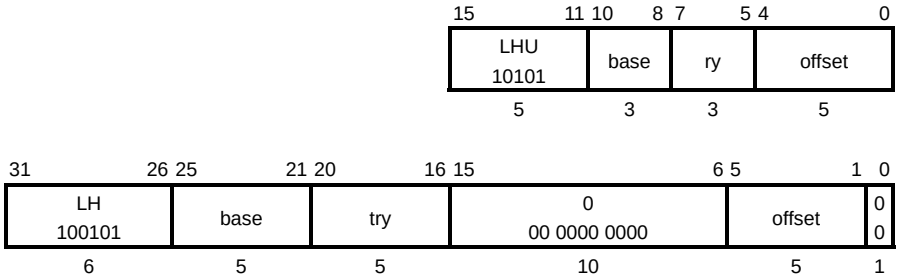
LHU *ry*, *offset* (*base*)

Load Halfword Unsigned

Operation

ry ⇐ {*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 EA is zero-extended and loaded into general-purpose register *ry*.

The *offset* field is 5 bits in length. Shifted one bit, this gives a range of 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

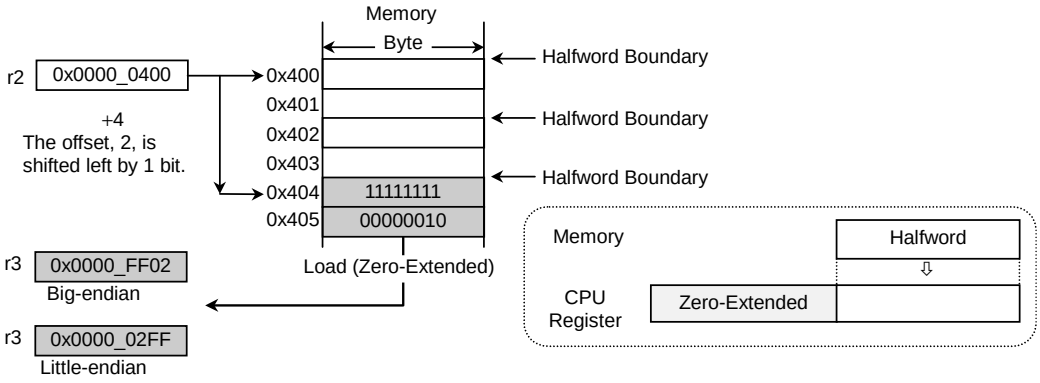
Address Error exception

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 MIPS16 decompressor, 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.


Memory
Byte
r2
0x0000_0400
0x400
0x401
0x402
0x403
0x404
0x405
11111111
00000010
Halfword Boundary
Halfword Boundary
Halfword Boundary
+4
The offset, 2, is
shifted left by 1 bit.
r3
0x0000_FF02
Big-endian
r3
0x0000_02FF
Little-endian
Load (Zero-Extended)
Memory
Halfword
CPU
Register
Zero-Extended

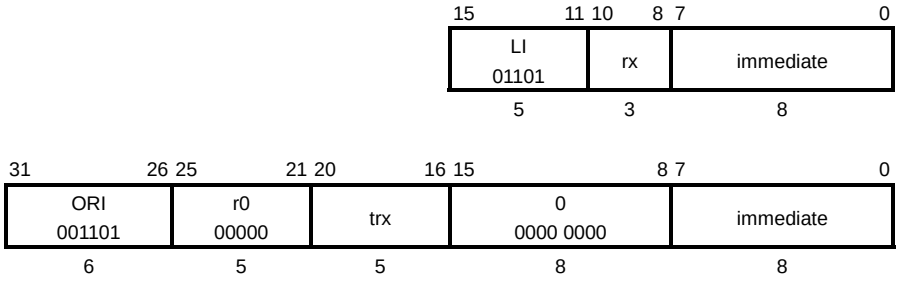
LI *rx*, *immediate*

Load Immediate

Operation

rx ⇐ *immediate*

Instruction Encoding



Description

The 8-bit *immediate* is zero-extended and loaded into general-purpose register *rx*.

The *immediate* field is 8 bits in length. This gives a range of 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.

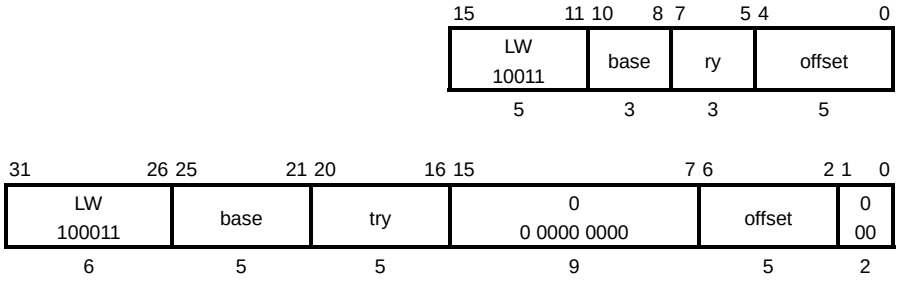
LW *ry*, *offset* (*base*)

Load Word

Operation

ry ⇐ (*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 EA is loaded into general-purpose register *ry*.

The *offset* field is 5 bits in length. Shifted two bits, this gives a range of 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

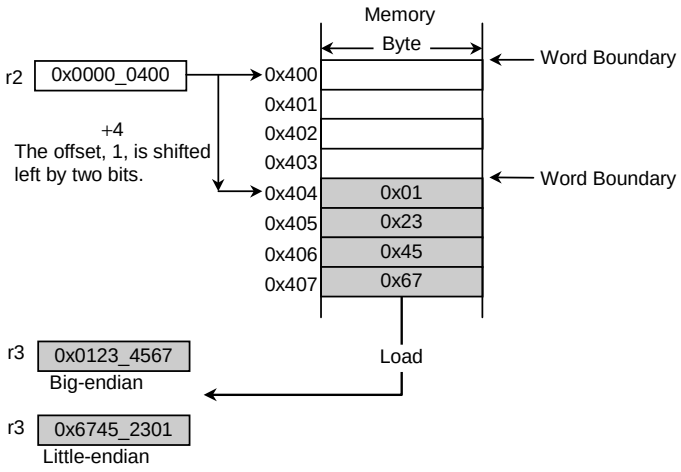
Address Error exception

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, 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.


Memory
Byte
Word Boundary
r2
0x0000_0400
0x400
0x401
0x402
0x403
+4
The offset, 1, is shifted
left by two bits.
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
Word Boundary
r3
0x0123_4567
Big-endian
Load
r3
0x6745_2301
Little-endian

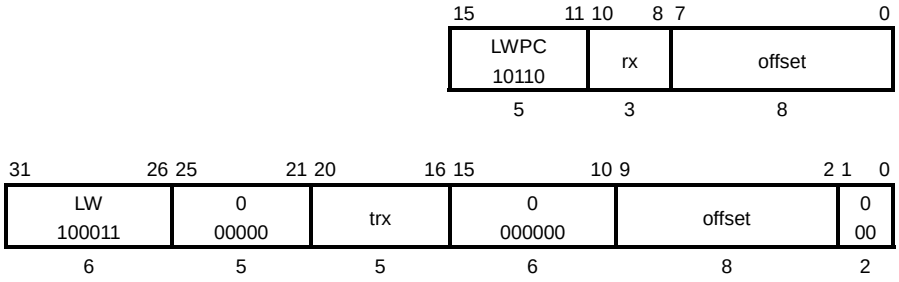
LW *rx*, *offset* (pc)

Load Word

Operation

rx ⇐ {*offset* (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 EA is then loaded into general-purpose register *rx*.

By virtue of this instruction, 32-bit constants can be embedded in the code segment. Instructions within the nearby routines can reference this data with a single instruction.

Zeros are shown in the field of bits 25 to 21 as placeholders. Because the LW instruction in the 32-bit ISA can not use the PC as the base register, the operation of this instruction differs from the LW instruction in the 32-bit ISA.

The *offset* field is 8 bits in length. Shifted two bits, this gives a range of 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.

	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

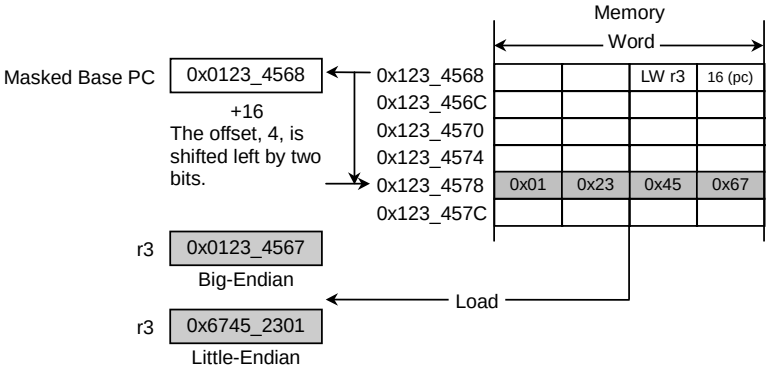
Address error exception

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 MIPS16 decompressor. 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.



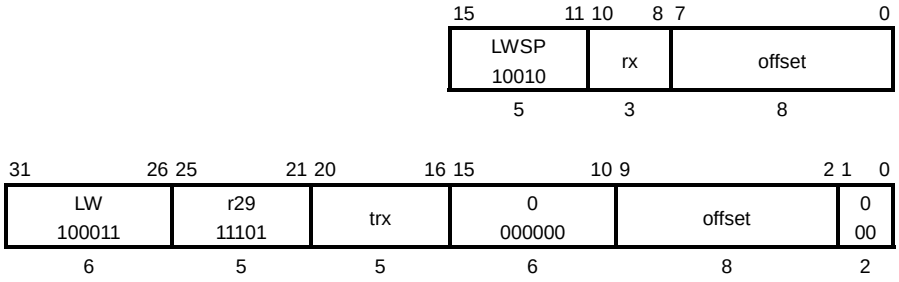
LW *rx*, *offset* (sp)

Load Word

Operation

rx ⇐ {*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 EA is loaded into general-purpose register *rx*.

The *offset* field is 8 bits in length. Shifted two bits, this gives a range of 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

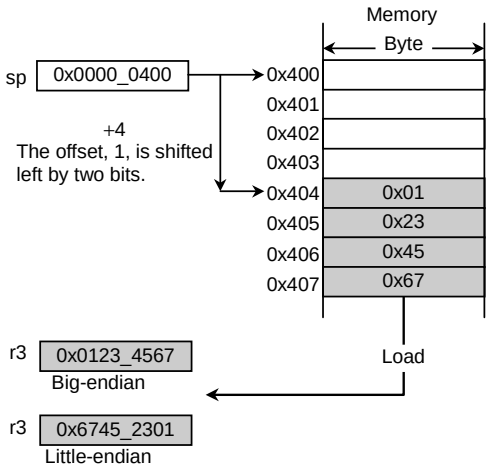
Address Error exception

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 MIPS16 decompressor. 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_23_01 in little-endian mode.


Memory
Byte
sp
0x0000_0400
0x400
0x401
0x402
0x403
+4
The offset, 1, is shifted
left by two bits.
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
r3
0x0123_4567
Big-endian
Load
r3
0x6745_2301
Little-endian

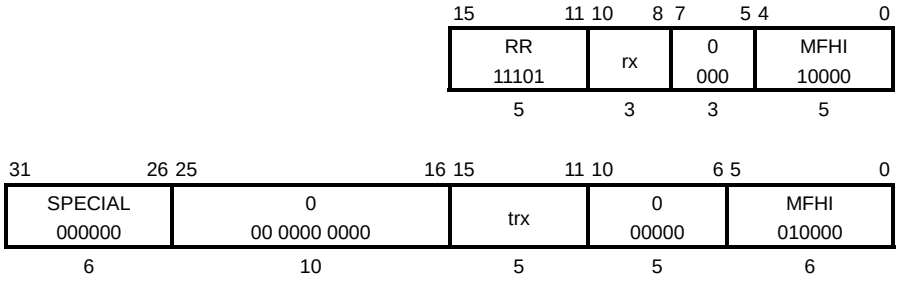
MFHI *rx*

Move From HI

Operation

rx ⇐ HI

Instruction Encoding



Description

The contents of the HI register is loaded into general-purpose register *rx*.

Exceptions

None

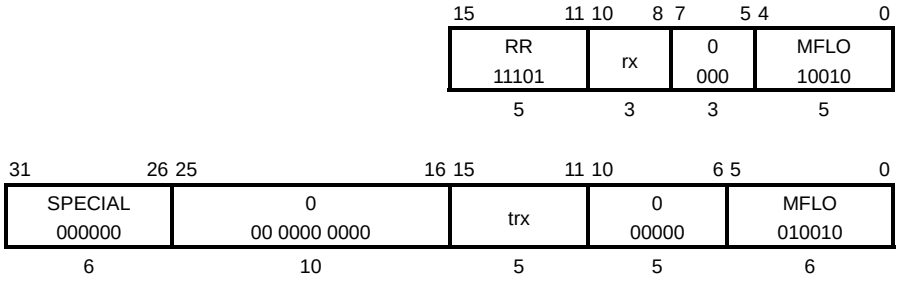
MFLO *rx*

Move From LO

Operation

rx ⇐ LO

Instruction Encoding



Description

The contents of the LO register is loaded into general-purpose register *rx*.

Exceptions

None

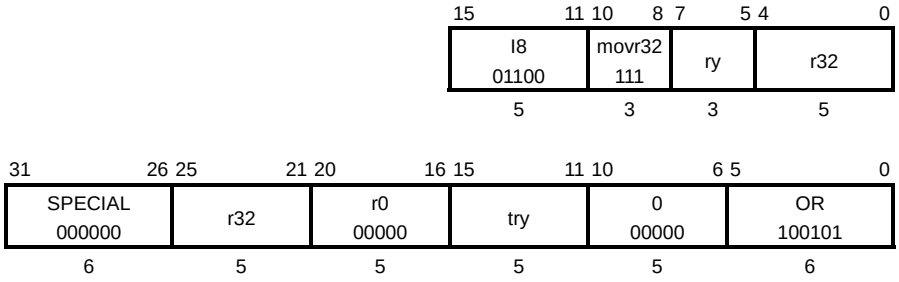
MOVE *ry*, *r32*

Move

Operation

ry ⇐ *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 identical to that of the 32-bit instructions; that is, 00000 is r0, 00001 is r1, 00010 is r2, 00011 is r3 and so on.

Exceptions

None

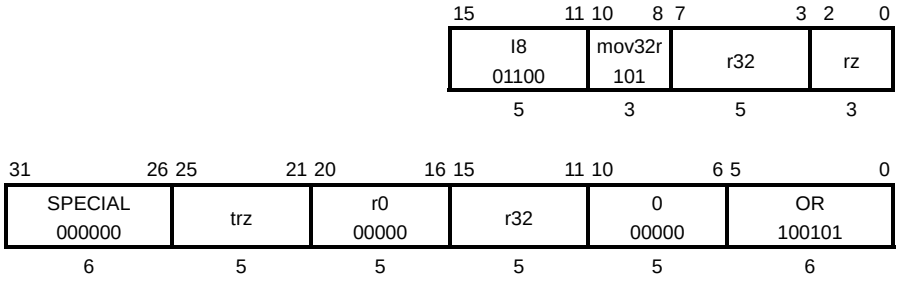
MOVE *r32*, *rz*

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

MULT *rx*, *ry*

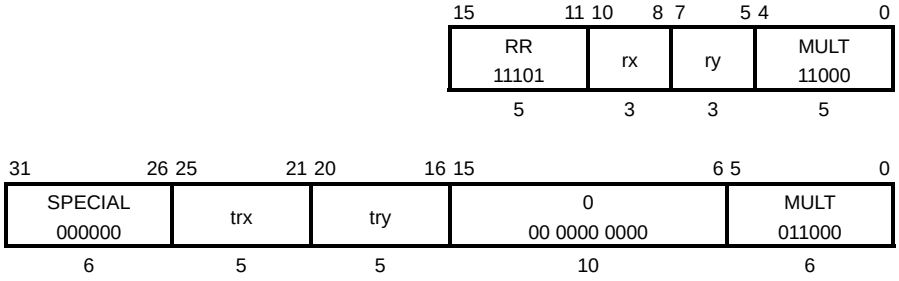
Multiply

Operation

HI ⇐ high-order word of (*rx* × *ry*);

LO ⇐ 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 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*

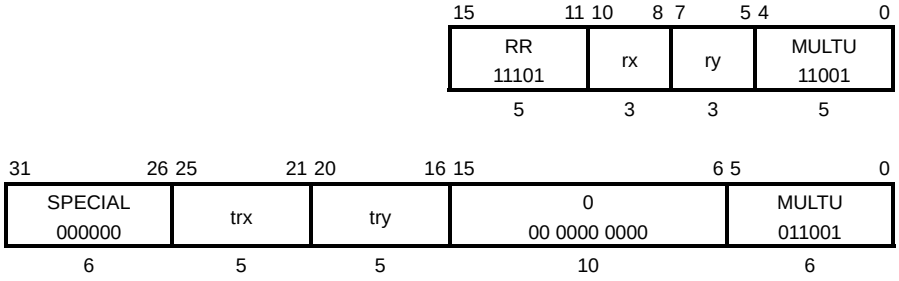
Multiply Unsigned

Operation

HI ⇐ high-order word of (*rx* × *ry*);

LO ⇐ 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.

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.

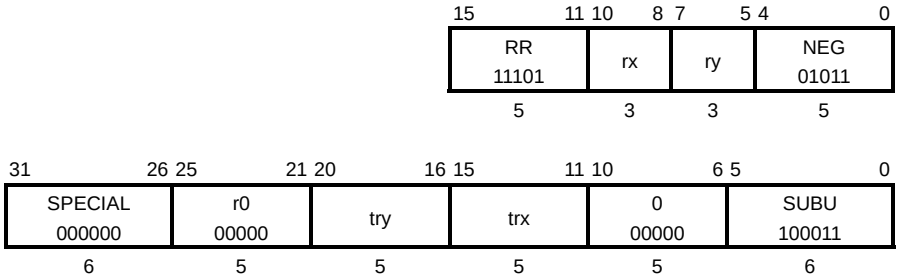
NEG *rx*, *ry*

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

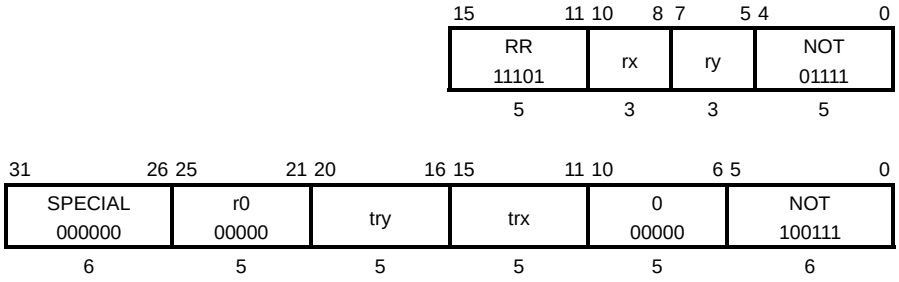
NOT *rx*, *ry*

NOT

Operation

rx ⇐ *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

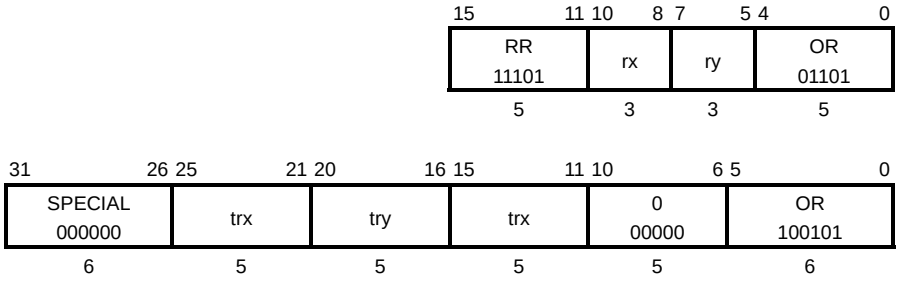
OR *rx*, *ry*

OR

Operation

rx ⇐ *rx* 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

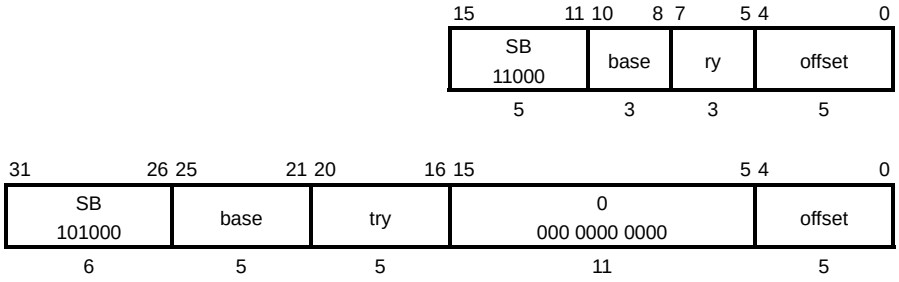
SB *ry*, *offset* (*base*)

Store Byte

Operation

ry ⇒ {*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 EA.

The three high-order bytes in *ry* is simply ignored; so there is no distinction between signed and unsigned stores.

The *offset* field is 5 bits in length. This gives a range of 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

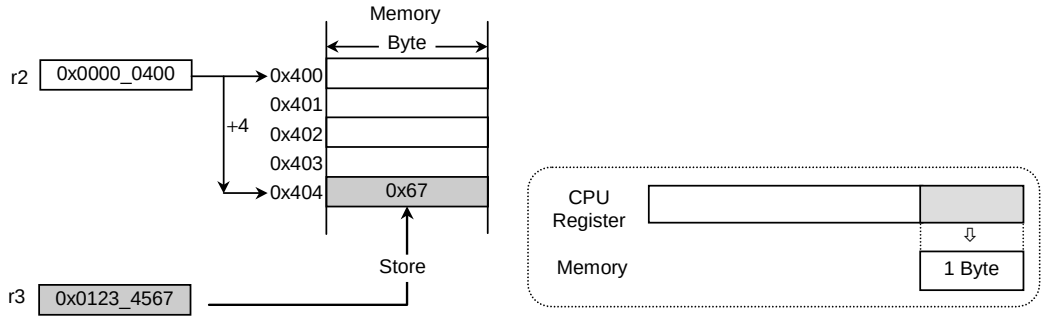
Address Error exception

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.


Memory
Byte
r2
0x0000_0400
0x400
0x401
0x402
0x403
0x404
+4
0x67
Store
r3
0x0123_4567
CPU
Register
Memory
1 Byte

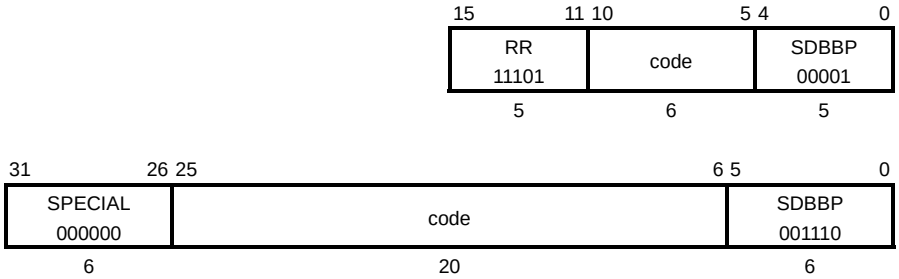
SDBBP *code*

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 while a Debug exception is being serviced (i.e., the DM bit in the Debug register is set). The operation of the SDBBP instruction is undefined when DM=1.

The SDBBP instruction may not be used within the user's program; it is intended for use by development systems.

Exceptions

Debug Breakpoint exception

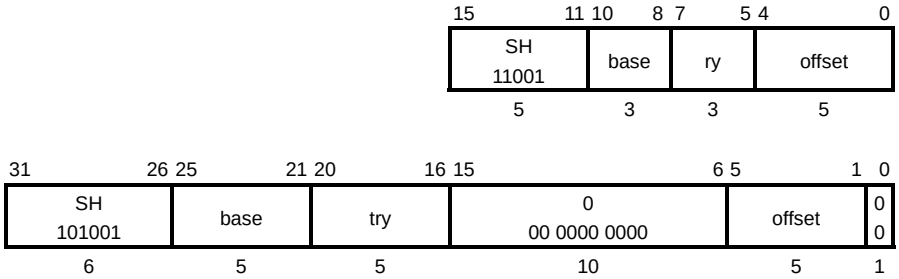
SH *ry*, *offset* (*base*)

Store Halfword

Operation

$ry \Rightarrow \{offset\ (base)\}$

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 least-significant halfword in general-purpose register *ry* is stored at the memory location addressed by EA.

The higher-order halfword in *ry* is simply ignored; so there is no distinction between signed and unsigned stores.

The *offset* field is 5 bits in length. Shifted one bit, this gives a range of 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

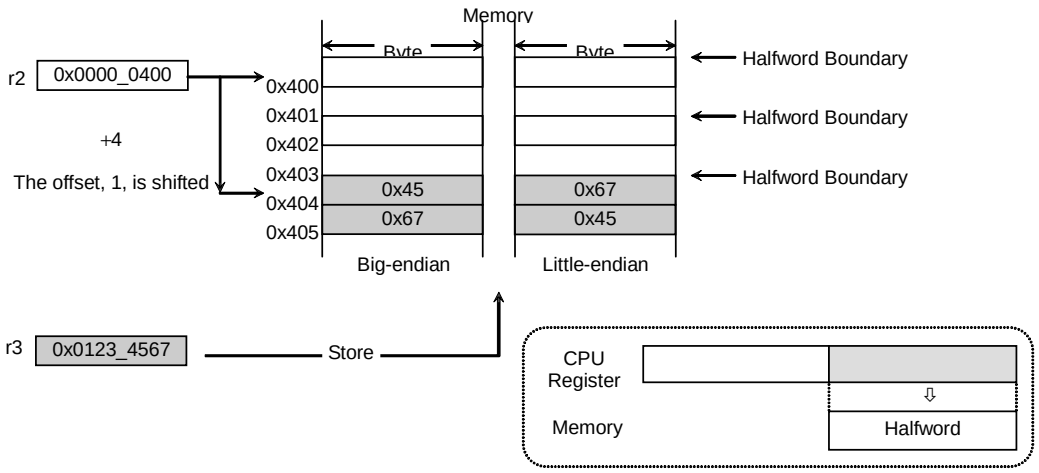
Address Error exception

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 MIPS16 decompressor, 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, this store instruction stores 0x45 and 0x67 to the memory locations at addresses 0x404 and 0x405 respectively. In little-endian mode, the above instruction stores 0x67 and 0x45 to the memory locations at addresses 0x404 and 0x405 respectively.



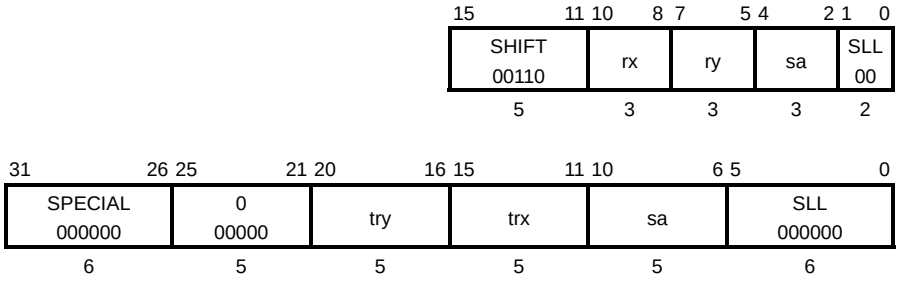
SLL *rx*, *ry*, *sa*

Shift Left Logical

Operation

$rx \Leftarrow ry << sa$

Instruction Encoding



Description

The 32-bit 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 general-purpose register *rx*. 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.

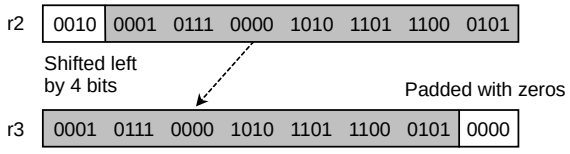
If the shift amount does not fit in the *sa* field, the instruction is EXTENDed to provide a full 5-bit field for a shift of 0 to 31.

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.



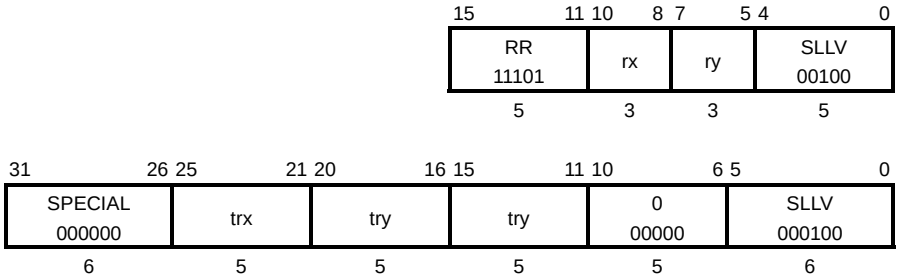
SLLV *ry*, *rx*

Shift Left Logical Variable

Operation

ry << 5 LSBs of *rx*

Instruction Encoding



Description

The 32-bit 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

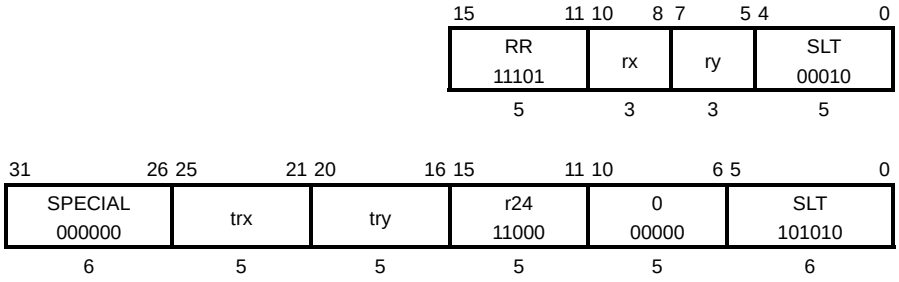
SLT *rx*, *ry*

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

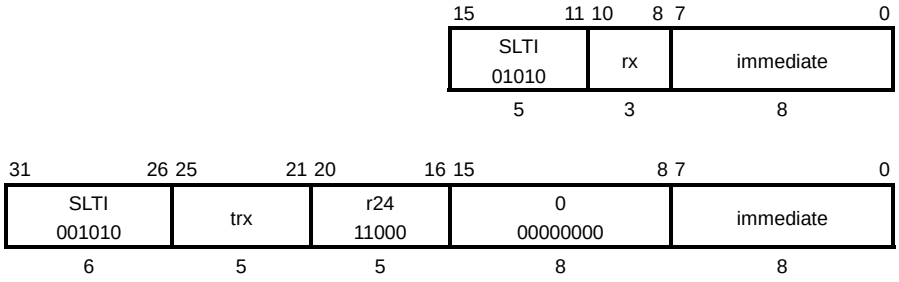
SLTI *rx*, *immediate*

Set On Less Than Immediate

Operation

if *rx* < *immediate* 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 the *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.

The *immediate* field is 8 bits in length. This gives a range of 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

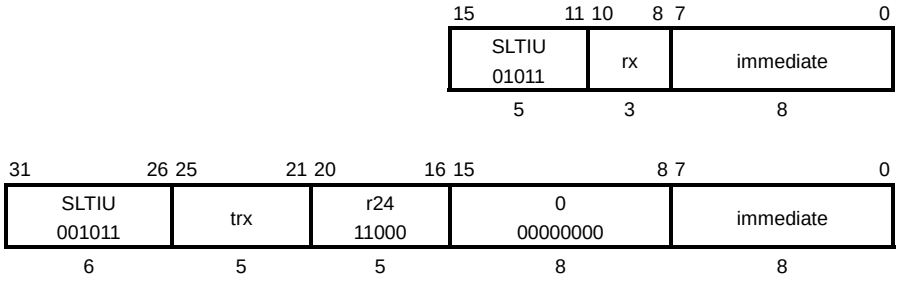
SLTIU *rx*, *immediate*

Set On Less Than Immediate Unsigned

Operation

if *rx* < *immediate* 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 unsigned integers. If *rx* is less than the 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.

The *immediate* field is 8 bits in length. This gives a range of 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

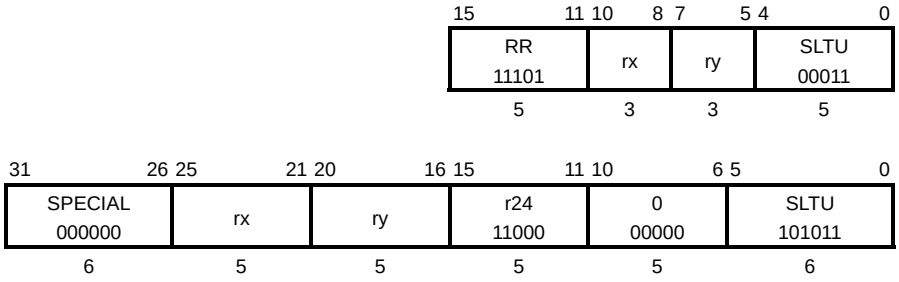
SLTU *rx*, *ry*

Set On Less Than Unsigned

Operation

if *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 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

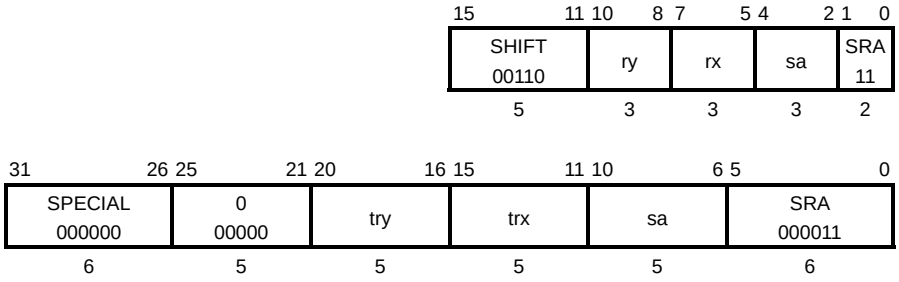
SRA *rx*, *ry*, *sa*

Shift Right Arithmetic

Operation

$rx \Leftarrow ry >> sa$

Instruction Encoding



Description

The 32-bit 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 general-purpose register *ry*. 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.

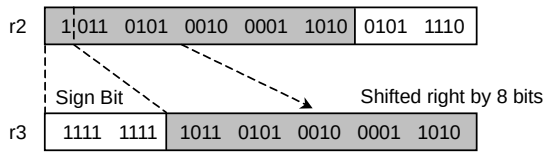
If the shift amount does not fit in the *sa* field, the instruction is EXTENDed to provide a full 5-bit field for shift of 0 to 31.

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.



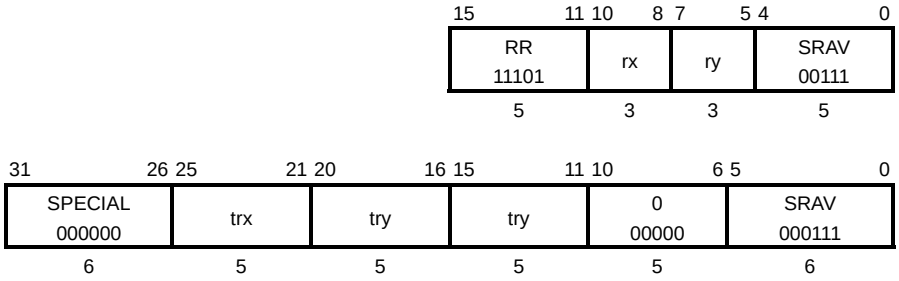
SRAV *ry*, *rx*

Shift Right Arithmetic Variable

Operation

ry >> 5 LSBs of *rx*

Instruction Encoding



Description

The 32-bit 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

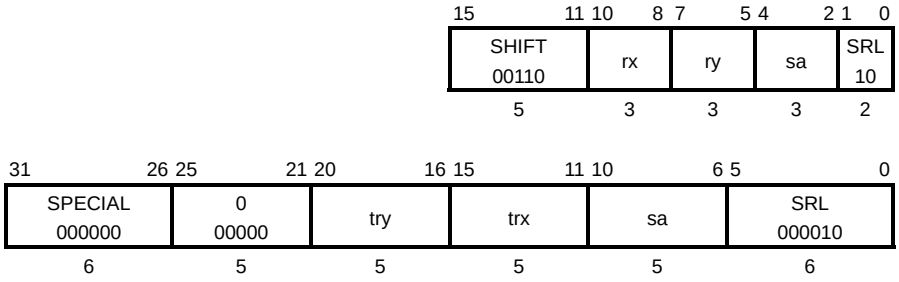
SRL *rx*, *ry*, *sa*

Shift Right Logical

Operation

$rx \Leftarrow ry >> sa$

Instruction Encoding



Description

The 32-bit 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 general-purpose register *rx*. 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.

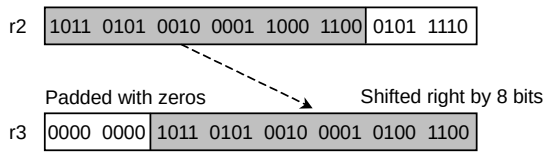
If the shift amount does not fit in the *sa* field, the instruction is EXTENDed to provide a full 5-bit field for a shift of 0 to 31.

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.



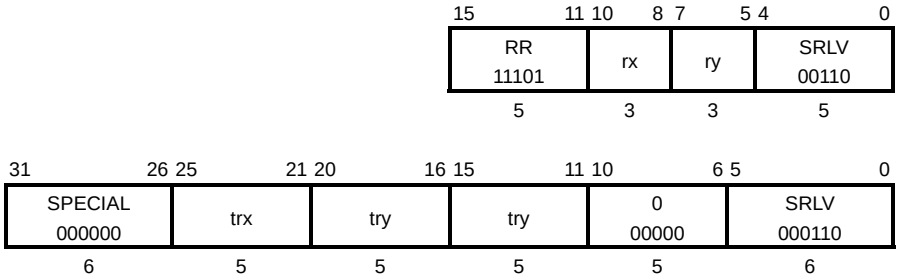
SRLV *ry*, *rx*

Shift Right Logical Variable

Operation

ry >> 5 LSBs of *rx*

Instruction Encoding



Description

The 32-bit 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*.

Exceptions

None

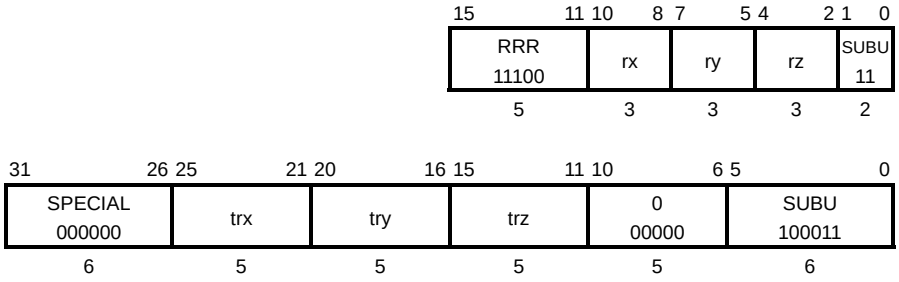
SUBU *rz*, *rx*, *ry*

Subtract Unsigned

Operation

$rz \Leftarrow rx - ry$

Instruction Encoding



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 overflow exception occurs under any circumstances.

Exceptions

None

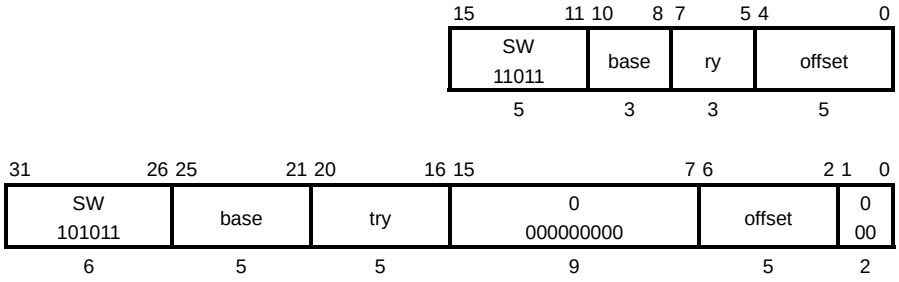
SW *ry*, *offset* (*base*)

Store Word

Operation

$ry \Rightarrow \{offset\ (base)\}$

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 EA.

The *offset* field is 5 bits in length. Shifted two bits, this gives a range of 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

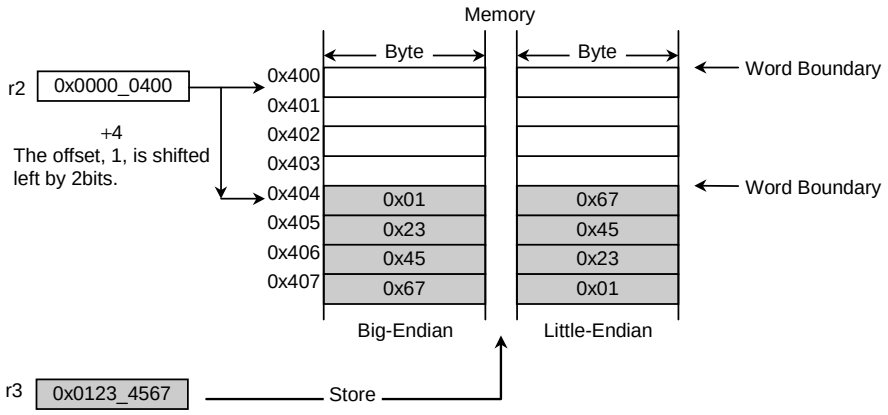
Address Error exception

Example

```
SW r3,4(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 MIPS16 decompressor, 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, this store instruction stores 0x12, 0x23, 0x45 and 0x67 to the memory locations at addresses 0x404 to 0x407 respectively. In little-endian mode, the above instruction stores 0x67, 0x45, 0x23 and 0x01 at addresses 0x404 to 0x407 respectively.


Memory
Byte
Byte
Word Boundary
r2
0x0000_0400
0x400
0x401
0x402
0x403
+4
The offset, 1, is shifted
left by 2bits.
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
0x67
0x45
0x23
0x01
Word Boundary
Big-Endian
Little-Endian
r3
0x0123_4567
Store

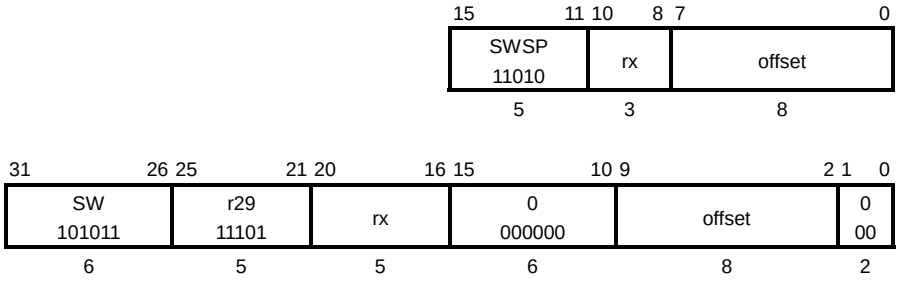
SW *rx*, *offset* (sp)

Store Word

Operation

$rx \Rightarrow \{offset\ (\text{sp})\}$

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 *rx* is stored at the memory location addressed by EA.

The *offset* field is 8 bits in length. Shifted two bits, this gives a range of 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. When EXTENDed, the *offset* operand is not shifted at all.

Exceptions

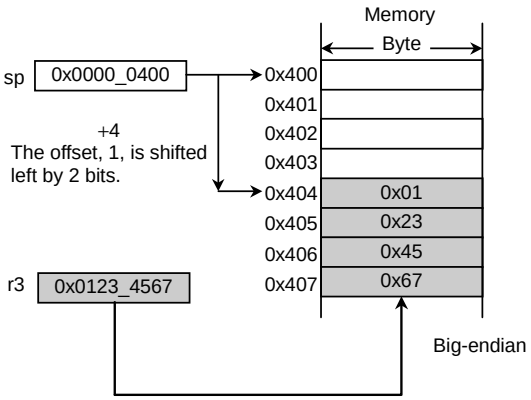
Address Error exception

Example

```
SW r3,4(sp)
```

Assume that registers sp and r3 contain 0x0000_0400 and 0x0123_4567 respectively. Since the offset value is shifted left by two bits by the processor, 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 0xD301.

In big-endian mode, this store instruction stores 0x1234_4567 to the memory locations at addresses 0x404 to 0x407 respectively.


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
r3
0x0123_4567
Big-endian

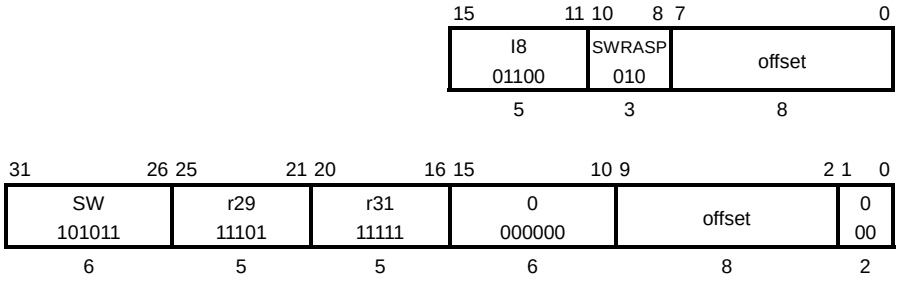
SW ra, *offset* (sp)

Store Word

Operation

ra ⇒ {*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 EA.

The *offset* field is 8 bits in length. Shifted two bits, this gives a range of 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. When EXTENDed, the *offset* operand is not shifted at all.

Exceptions

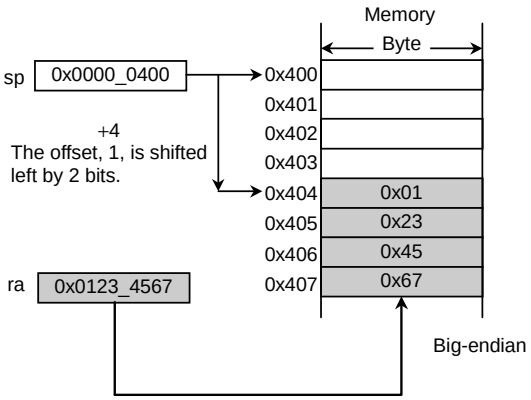
Address Error exception

Example

```
SW ra,4(sp)
```

Assume that registers sp and ra contain 0x0000_0400 and 0x0123_4567 respectively. Since the offset value is shifted left by two bits by the processor, 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 0x3101.

In big-endian mode, this store instruction stores 0x1234_4567 to the memory locations at addresses 0x404 to 0x407 respectively.


Memory
Byte
sp
0x0000_0400
0x400
0x401
0x402
0x403
+4
The offset, 1, is shifted
left by 2 bits.
0x404
0x01
0x405
0x23
0x406
0x45
ra
0x0123_4567
0x407
0x67
Big-endian

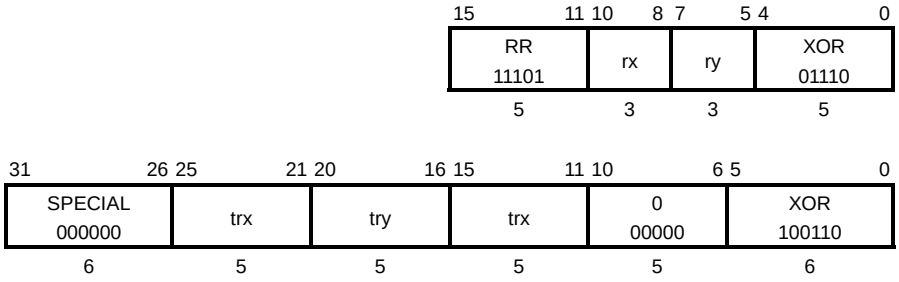
XOR *rx*, *ry*

Exclusive OR

Operation

rx ⇐ *rx* 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

Appendix C Programming Restrictions

In a pipelined machine like the TX19, there are certain instructions which due to the very pipeline structure could disrupt the smooth operation of the pipeline. 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		Restriction
LH	*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.
LHU	*rt*, *offset*(*base*)	
SH	*rt*, *offset*(*base*)	
LW	*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.
LWU	*rt*, *offset*(*base*)	
SW	*rt*, *offset*(*base*)	

Table C-2 Jump Instructions

Instructions		Restriction
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	Restriction
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	The branch instruction may not be in a jump or branch delay slot. The operation of the branch instruction is undefined if it is in a jump or branch delay slot.

Table C-4 System Control Coprocessor (CP0) Instructions

Instructions	Restriction
CACHE *op*, *offset*(*base*) DERET MTC0 *rt*, *rd* MFC0 *rt*, *rd* RFE	Attempts by a User-mode program to execute these instructions when the CU[0] 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 CU[0] bit.
DERET	• The NOP instruction must be inserted in the delay slot following this instruction. • 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. • This instruction must not be executed immediately after the MTC0 instruction that writes to the Debug register or immediately after the MFC0 instruction that reads from the Debug register. Otherwise, the contents of the Debug register become undefined.
MTC0 *rt*, *rd*	• The MTC0 instruction may not attempt to write to the Status register immediately before the RFE instruction. Otherwise, the contents of the Status register become undefined. • The MTC0 instruction may not attempt to write to the Debug register immediately before the DERET instruction. Otherwise, the contents of the Debug register become undefined.
MFC0 *rt*, *rd*	• The MFC0 instruction may not attempt to read the Status register immediately before the RFE instruction. Otherwise, the contents of the Status register become undefined. • The MFC0 instruction may not attempt to read the Debug register immediately before the DERET instruction. Otherwise, the contents of the Debug register become undefined. • The MFC0 instruction has a delay slot.
RFE	• This instruction may not be executed immediately after an MTC0 instruction that writes to the Status register or immediately after an MFC0 instruction that reads from the Status register. Otherwise, the contents of the Status register become undefined. • The contents of the Status register become unpredictable if an interrupt occurs during execution of the RFE instruction. Therefore, all interrupts must be disabled prior to the RFE instruction.

Table C-5 Coprocessor Instructions

Instructions		Restriction
BC*z*F(L)	*offset*	Attempted execution of these instructions causes a Coprocessor Unusable exception when the corresponding CU bit in the Status register is cleared.
BC*z*T(L)	*offset*	
CFC*z*	*rt*, *rd*	
CTC*z*	*rt*, *rd*	
COP*z*	*cofun*	
MFC*z*	*rt*, *rd*	
MTC*z*	*rt*, *rd*	

Table C-6 Special Instructions

Instructions	Restriction
SDBBP	This instruction may not be executed while a Debug exception is being serviced (i.e., the DM bit in the Debug register is set). The operation of the SDBBP instruction is undefined when DM=1.

C.2 16-Bit ISA Restrictions

Table C-7 Load and Store Instructions

Instructions		Restriction
LH	*ry*, *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.
LHU	*ry*, *offset*(*base*)	
SH	*ry*, *offset*(*base*)	
LW	*ry*, *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.
SW	*ry*, *offset*(*base*)	
SW	*ry*, *offset*(sp)	
SW	*ra*, *offset*(sp)	

Table C-8 Jump Instructions

Instructions	Restriction
JALR 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*	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	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 instruction may not be in a jump delay slot.

Table C-9 Branch Instructions

Instructions	Restriction
All branch instructions	The branch instruction may not be in a jump delay slot.

Table C-10 Special Instructions

Instructions	Restriction
SDBBP	This instruction may not be executed while a Debug exception is being serviced (i.e., the DM bit in the Debug register is set). The operation of the SDBBP instruction is undefined when DM=1.

Table C-11 EXTENDed Instructions

Instructions	Restriction
All EXTENDed instructions	Any EXTENDed instruction may not be in a jump delay slot.

Appendix D Compatibility Among TX19, TX39 and R3000A Architectures

Table D-1 shows the differences between Toshiba's TX19 and TX39.

Table D-1 Comparisons Between the TX19 and the TX39

Feature	TX19		TX39
Application	Low power, high code density		High performance
Instruction Set	32-Bit ISA	16-Bit ISA	• 32-bit fixed instruction size • 84 instructions
	• Object-code compatible upward from the TX39 • 85 instructions, including JALX for run-time switching between ISA modes	• Object-code compatible with the MIPS16 ASE except doubleword and LWU instructions. • 58 instructions	
CPU Registers	• Same as for the TX39 • The least-significant bit of the PC determines the ISA mode.		• 32 general-purpose registers • Program counter (PC) • 2 Multiply/Divide registers (HI/LO) All CPU registers are 32-bits wide.

Feature	TX19	TX39
CP0 Registers	The new Interrupt Enable (IE) register provides for single-instruction enabling/disabling of interrupts.	• 1 system configuration register • 6 general exception handling registers • 2 debug exception handling registers All CP0 registers are 32-bits wide.
	The definitions of the following register bits differ between the TX19 and the TX39: PRId[15:8] Implementation=0x2C Cause[11:8] Sw Cause[15:13] IL Status[11:8] SWiMask Status[15:13] CMask Status[18:16] PMask Status[25] Reserved	PRId[15:8] Implementation=0x22 Cause[9:8] Sw Cause[15:10] IP Status[9:8] IntMask (Sw) Status[15:10] IntMask (Int) Status[18:16] 0 Status[25] RE
Instruction Pipeline	5-stage	5-stage
Multiply Instructions	Latency / Execution = 2 / 1cycles	Latency / Execution = 2 / 1 cycles
Divide Instructions	Latency / Execution = 35 / 34 cycles If the divide instruction is followed by a Move From HI/LO instruction before the result is made available, the pipeline stalls until the result does become available.	Latency / Execution = 35 / 34 cycles If the divide instruction is followed by a Move From HI/LO instruction before the result is made available, the divide instruction is canceled.
Multiply-and-Add Instructions	Latency / Execution = 2 / 1 cycles	Latency / Execution = 2 / 1 cycles
Interrupt Response	• Interrupt requests are processed by hardware. • Exceptions and interrupts have distinct vector addresses. • The interrupt mask level is automatically updated by hardware. • Optional on-chip RAM provides for an interrupt stack with a single-clock access.	• Interrupt requests are processed by software. • Exceptions and interrupts have a common vector address. • The interrupt mask level needs to be updated under software control.
Maskable Interrupts	• 4 software interrupts • 1 hardware interrupt from the interrupt controller (7 prioritized levels)	• 2 software interrupts • 6 hardware interrupts
Virtual Address Space	4 Gbytes	4 Gbytes
Clock Rate	20 MHz (standard version), A high-speed version is being planned.	70 MHz

Table C-2 gives comparisons of the instruction sets for the TX19 (32-bit ISA), the TX39 and the MIPS R3000A. Differences are highlighted in shaded boxes.

Table D-2 Instruction Sets of the TX19, the TX39 and the R3000A

Category	Instruction	TX19 32-Bit ISA		TX39		R3000A	
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*)
	Sync	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*, *immediate*	LUI	*rt*, *immediate*	LUI	*rt*, *immediate*
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*

Category	Instruction	TX19 32-Bit ISA		TX39		R3000A	
Shift	Shift Left Logical	SLL	*rd*, *rt*, *sa*	SLL	*rd*, *rt*, *sa*	SLL	*rd*, *rt*, *sa*
	Shift Left Logical Variable	SLLV	*rd*, *rt*, *rs*	SLLV	*rd*, *rt*, *rs*	SLLV	*rd*, *rt*, *rs*
	Shift Right Logical	SRL	*rd*, *rt*, *sa*	SRL	*rd*, *rt*, *sa*	SRL	*rd*, *rt*, *sa*
	Shift Right Logical Variable	SRLV	*rd*, *rt*, *rs*	SRLV	*rd*, *rt*, *rs*	SRLV	*rd*, *rt*, *rs*
	Shift Right Arithmetic	SRA	*rd*, *rt*, *sa*	SRA	*rd*, *rt*, *sa*	SRA	*rd*, *rt*, *sa*
	Shift Right Arithmetic Variable	SRAV	*rd*, *rt*, *rs*	SRAV	*rd*, *rt*, *rs*	SRAV	*rd*, *rt*, *rs*
Multiply and Divide	Multiply	MULT	*rs*, *rt*	MULT	*rs*, *rt*	MULT	*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*		
	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	Multiply-and-Add	MADD	*rs*, *rt*	MADD	*rs*, *rt*		
		MADD	*rd*, *rs*, *rt*	MADD	*rd*, *rs*, *rt*		
	Multiply-and-Add Unsigned	MADDU	*rs*, *rt*	MADDU	*rs*, *rt*		
		MADDU	*rd*, *rs*, *rt*	MADDU	*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*				
	Jump Register	JR	*rs*	JR	*rs*	JR	*rs*
	Jump And Link Register	JALR	(*rd*,) *rs*	JALR	*(rd,) rs*	JALR	*(rd,) rs*
Branch	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 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 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	TX19 32-Bit ISA	TX39	R3000A
Branch-Likely	Branch On Equal Likely	BEQL *rs*, *rt*, *offset*	BEQL *rs*, *rt*, *offset*	
	Branch On Not Equal Likely	BNEL *rs*, *rt*, *offset*	BNEL *rs*, *rt*, *offset*	
	Branch On Greater Than Zero Likely	BGTZL *rs*, *offset*	BGTZL *rs*, *offset*	
	Branch On Greater Than or Equal to Zero Likely	BGEZL *rs*, *offset*	BGEZL *rs*, *offset*	
	Branch On Less Than Zero Likely	BLTZL *rs*, *offset*	BLTZL *rs*, *offset*	
	Branch On Less Than or Equal to Zero Likely	BLEZL *rs*, *offset*	BLEZL *rs*, *offset*	
	Branch On Less Than Zero And Link Likely	BLTZALL *rs*, *offset*	BLTZALL *rs*, *offset*	
	Branch On Greater Than Zero And Link Likely	BGEZALL *rs*, *offset*	BGEZALL *rs*, *offset*	
Coprocessor	Move To Coprocessor	MTC*z* *rt*, *rd*	MTC*z* *rt*, *rd*	MTC*z* *rt*, *rd*
	Move From Coprocessor	MFC*z* *rt*, *rd*	MFC*z* *rt*, *rd*	MFC*z* *rt*, *rd*
	Move Control To Coprocessor	CTC*z* *rt*, *rd*	CTC*z* *rt*, *rd*	CTC*z* *rt*, *rd*
	Move Control From Coprocessor	CFC*z* *rt*, *rd*	CFC*z* *rt*, *rd*	CFC*z* *rt*, *rd*
	Coprocessor Operation	COP*z* *cofun*	COP*z* *cofun*	COP*z* *cofun*
	Branch On Coprocessor *z* True	BC*z*T *offset*	BC*z*T *offset*	BC*z*T *offset*
	Branch On Coprocessor *z* True Likely	BC*z*TL *offset*	BC*z*TL *offset*	BC*z*TL *offset*
	Branch On Coprocessor *z* False	BC*z*F *offset*	BC*z*F *offset*	BC*z*F *offset*
	Branch On Coprocessor *z* False Likely	BC*z*FL *offset*	BC*z*FL *offset*	BC*z*FL *offset*
	Load Word To Coprocessor			LWC*z* *rt*, *offset*(*base*)
	Store Word From Coprocessor			SWC*z* *rt*, *offset*(*base*)
System Control Coprocessor	Move To CP0	MTC0 *rt*, *rd*	MTC0 *rt*, *rd*	
	Move From CP0	MFC0 *rt*, *rd*	MFC0 *rt*, *rd*	
	Restore From Exception	RFE	RFE	
	Debug Exception Return	DERET	DERET	
	Cache	CACHE *op*, *offset*(*base*)	CACHE *op*, *offset*(*base*)	
	Read Indexed TLB Entry†	(TLBR)	(TLBR)	TLBR
	Write Indexed TLB Entry†	(TLBWI)	(TLBWI)	TLBWI
	Write Random TLB Entry†	(TLBWR)	(TLBWR)	TLBWR
	Probe TLB For Matching Entry†	(TLBP)	(TLBP)	TLBP

Category	Instruction	TX19 32-Bit ISA	TX39	R3000A
Special	System Call	SYSCALL *code*	SYSCALL *code*	SYSCALL *code*
	Breakpoint	BREAK *code*	BREAK *code*	BREAK *code*
	Software Debug Breakpoint Exception	SDBBP *code*	SDBBP *code*	

† No operation is performed in the TX19 and the TX39L.

Table C-3 gives comparisons of the instruction sets supported by the TX19 16-bit ISA mode and the MIPS16 ASE. The TX19 is object-code compatible with the MIPS16 ASE except that the doubleword instructions plus the Load Word Unsigned (LWU) instruction are not implemented in the TX19.

Table D-3 Instruction Sets of the TX19 16-bit ISA and the MIPS16 ASE

Category	Instruction	TX19 16-Bit ISA	MIPS16 ASE
Load and Store	Load Byte	LB *ry*, *offset*(*base*)	LB *ry*, *offset*(*base*)
	Load Byte Unsigned	LBU *ry*, *offset*(*base*)	LBU *ry*, *offset*(*base*)
	Load Halfword	LH *ry*, *offset*(*base*)	LH *ry*, *offset*(*base*)
	Load Halfword Unsigned	LHU *ry*, *offset*(*base*)	LHU *ry*, *offset*(*base*)
	Load Word	LW *ry*, *offset*(*base*)	LW *ry*, *offset*(*base*)
		LW *ry*, *offset*(pc)	LW *ry*, *offset*(pc)
		LW *ry*, *offset*(sp)	LW *ry*, *offset*(sp)
	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*)
	Store Halfword	SH *ry*, *offset*(*base*)	SH *ry*, *offset*(*base*)
	Store Word	SW *ry*, *offset*(*base*)	SW *ry*, *offset*(*base*)
		SW *ry*, *offset*(pc)	SW *ry*, *offset*(pc)
		SW *ry*, *offset*(sp)	SW *ry*, *offset*(sp)
	Store Doubleword		SD *ry*, *offset*(*base*)
			SD *ry*, *offset*(pc)
			SD *ry*, *offset*(sp)
ALU Immediate	Add Immediate	ADDIU *ry*, *rx*, *immediate*	ADDIU *ry*, *rx*, *immediate*
		ADDIU *rx*, *immediate*	ADDIU *rx*, *immediate*
		ADDIU sp, *immediate*	ADDIU *sp, immediate*
		ADDIU *rx*, pc, *immediate*	ADDIU *rx*, pc, *immediate*
		ADDIU *rx*, sp, *immediate*	ADDIU *rx*, sp, *immediate*
	Doubleword Add Immediate		DADDIU *ry*, *rx*, *immediate*
			DADDIU *ry*, *immediate*
			DADDIU *ry*, sp, *immediate*
			DADDIU sp, *immediate*
			DADDIU *ry*, pc, *immediate*
	Set On Less Than Immediate	SLTI *rx*, *immediate*	SLTI *rx*, *immediate*
	Set On Less Than Immediate Unsigned	SLTIU *rx*, *immediate*	SLTIU *rx*, *immediate*
	Compare Immediate	CMPI *rx*, *immediate*	CMPI *rx*, *immediate*
	Load Immediate	LI *rx*, *immediate*	LI *rx*, *immediate*

Category	Instruction	TX19 16-Bit ISA	MIPS16 ASE
2/3-Operand Register-Type	Add Unsigned	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*
	Doubleword Subtract Unsigned		DSUBU *rz*, *rx*, *ry*
	Set On Less Than	SLT *rx*, *ry*	SLT *rx*, *ry*
	Set On Less Than Unsigned	SLTU *rx, ry*	SLTU *rx, ry*
	Compare	CMP *rx, ry*	CMP *rx, ry*
	Negate	NEG *rx, ry*	NEG *rx, ry*
	AND	AND *rx, ry*	AND *rx, ry*
	OR	OR *rx, ry*	OR *rx, ry*
	Exclusive-R	XOR *rx, ry*	XOR *rx, ry*
	Not	NOT *rx, ry*	NOT *rx, ry*
	Move	MOVE *ry*, *r32*	MOVE *ry*, *r32*
		MOVE *r32*, *rz*	MOVE *r32*, *rz*
Shift	Shift Left Logical	*SLL* *rx*, *ry*, *sa*	SLL *rx*, *ry*, *sa*
	Shift Left Logical Variable	SLLV ry, *rx*	SLLV *ry, rx*
	Shift Right Logical	SRL *rx*, *ry*, *sa*	SRL *rx*, *ry*, *sa*
	Shift Right Logical Variable	SRLV *ry*, *rx*	SRLV *ry*, *rx*
	Shift Right Arithmetic	SRA *rx*, *ry*, *sa*	SRA *rx*, *ry*, *sa*
	Shift Right Arithmetic Variable	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 *ry*, *sa*
	Doubleword Shift Right Logical Variable		DSRLV *ry*, *rx*
	Doubleword Shift Right Arithmetic		DSRA *ry*, *sa*
	Doubleword Shift Right Arithmetic Variable		DSRAV *ry*, *rx*
Multiply and Divide	Multiply	MULT *rx, ry*	MULT *rx, ry*
	Multiply Unsigned	MULTU *rx, ry*	MULTU *rx, ry*
	Doubleword Multiply		DMULT *rx, ry*
	Doubleword Multiply Unsigned		DMULTU *rx, ry*
	Divide	DIV *rx, ry*	DIV *rx, ry*
	Divide Unsigned	DIVU *rx, ry*	DIVU *rx, ry*
	Doubleword Divide	DIV *rx, ry*	DDIV *rx, ry*
	Doubleword Divide Unsigned	DIVU *rx, ry*	DDIVU *rx, ry*
	Move From HI	MFHI *rx*	MFHI *rx*
	Move From LO	MFLO *rx*	MFLO *rx*

Category	Instruction	TX19 16-Bit ISA		MIPS16 ASE	
Jump	Jump And Link	JAL	*target*	JAL	*target*
	Jump And Link eXchange	JALX	*target*	JALX	*target*
	Jump Register	JR	*rx*	JR	*rx*
		JR	ra	JR	ra
	Jump And Link Register	JALR	ra, *rx*	JALR	*ra, rx*
Branch	Branch On Equal To Zero	BEQZ	*rx*, *offset*	BEQZ	*rx*, *offset*
	Branch On Not Equal To Zero	BNEZ	*rx*, *offset*	BNEZ	*rx*, *offset*
	Branch On T8 Equal To Zero	BTEQZ	*offset*	BTEQZ	*offset*
	Branch On T8 Not Equal to Zero	BTNEZ	*offset*	BTNEZ	*offset*
	Branch Unconditional	B	*offset*	B	*offset*
Special	Breakpoint	BREAK	*code*	BREAK	*code*
	Software Debug Breakpoint Exception	SDBBP	*code*	SDBBP	*code*
	Extend	EXTEND	*immediate*	EXTEND	*immediate*