MIPS32 4K™

Processor Core Family

Software User's Manual

Revision 01.07

June 19, 2000

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

:
_____

References to Product Names

This manual encompasses the 4Kc™, 4Kp™ & 4Km™ processor cores. The three products are similar in design, hence the majority of information contained in this manual refers to all three cores.

Throughout this manual the terms "the core" or "the processor" refers to the 4Kc™, 4Kp™, and 4Km™ devices. Some information in this manual, specifically in Chapters 2 and 4, is specific to one or more of the cores, but not all three. This information is called out in the text wherever necessary. For example, the section dealing with the TLB is denoted as being 4Kc™ core specific, whereas the section dealing with the BAT is denoted as being 4Kp™ and 4Km™ core specific.

Product Differentiation

The three products contained in this manual are similar in design. The main differences are in memory management and the multiply-divide unit. In general the differences are as follows:

- 4Kc™ processor: Contains pipelined multiplier and translation lookaside buffer (TLB).

- 4Kp™ processor: Contains non-pipelined multiplier and block address translator (BAT).

- 4Km™ processor: Contains pipelined multiplier and block address translator.

:

Revision History

Revision	Date	PrID Rev. Number	Description
1.0	August, 1999	0x01	First released version
1.1	November, 1999	0x02	- Re-organization to be more of a SoftWare User's Manual. Removed System Interface chapter. - Count register no longer stops incrementing in DebugMode - New bit added to Debug register to indicate this: CountDM - New Bits added to Debug register for handling of imprecise exceptions: IEXI, DBusEP, IBusEP - Added description of SubBlock ordering - New MDU timing. Updated pipeline diagrams and text in Chap. 2 to reflect new timing - Modified Reset description. SoftReset cannot be masked by the core. SoftReset does not need to be asserted when Reset is asserted - ASID is not used in EJTAG breakpoint comparisons if the TLB is not implemented - Added MT Compare to Timer Interrupt cleared to list of Hazard conditions - Fixed Hazard from setting of SW Interrupt to Interrupted instruction - Changed SPECIAL opcode map to reflect MOVCI FP instn as a Coprocessor Instn rather than a Reserved Instn - L2 Cache encodings of CACHE instn are reserved. - Added note that I Fill CACHE instn will cause a re-fetch even if the line is in the cache

Revision	Date	PrID Rev. Number	Description
1.1, continued	November, 1999	0x02	• MUL instn description reiterates that the contents of HI/LO are unpredictable after the MUL operation. • Added ERL=1 as possible reason for being in kernel mode in the kseg descriptions • Swapped priority of RI and CU exceptions • Changed general exception code pseudo-code to have correct vector offset of 0x180 • Fixed typo in bus error description: stores OR non-critical words... not stores of non-critical words • Changed TLBWI to TLBWR in Random register description • Added note that behavior is undefined if illegal page mask value is used • Added note that $Status_{TS}$, $Status_{SR}$, and $Status_{NMI}$ bits and $Cause_{WP}$ cannot be set by software • Noted undefined behavior if $Status_{ERL}$ is set while executing code in useg/kuseg • Added $Config1_{PC}$ and $Config1_{CA}$ bits. Both wired to 0 • Changed Reset state of $Watch_I$, $Watch_R$, and $Watch_W$ to 0 from undefined • Removed some false statements about WAIT induced sleep mode • CLO/CLZ instn description changed to reflect use of rd as destination register instead of rt • Add sel field to format statements in MFC0/MTC0 instns • Removed redundant statement about writeback invalidate from PREF instn • Add programming note to multiply instructions that smaller source value should be placed in rt • Updated listing of HW initialized Cop0 bits in Reset chapter

Revision	Date	PrID Rev. Number	Description
1.2	December, 1999	0x02	• Removed implication of internal mux for SI_TimerInt from description of Compare register
01.03	January 28, 2000	0x04	• Cleaned up old references to 'both' cores • Fixed some typos • Fixed pipe stages in figure 2-12 • Added details on D-side micro TLB • Cleaned up usage of trademarks • Renamed title to *MIPS32 4k™ Processor Core Family Software User's Manual* • Changed revision numbering to xx.yy format for consistency with other documents
01.04	March 23, 2000	0x05	• Cleaned up some old paragraph leftovers • Changed look of Table of Contents, List of Figures and List of Tables • Added timing information regarding Early In to divide algorithm for 4Kc and 4Km • Fixed CLO/CLZ description in section 10.7 to reflect rt -> rd change in definition • Cleaned up Config register definition. Defined BM field, defined reset state of several fields. Changed reserved fields to 0 fields • Cleaned up decode tables - fixed font problems and multi-line instn text • Updated PREF description • Made reset state of $Status_{RP}$ 0 • Fixed some Spell-check issues.
01.05	May 8, 2000	0x06	• Clarified "Fetch and Lock" CACHE description. • Removed text saying that the upper bits of PrID were available for implementors.
01.06	June 8, 2000	0x06	• Rephrased field description of DataLo register. • Updated copyright and trademark notices.

Revision	Date	PrID Rev. Number	Description
01.07	June 19, 2000	0x06	• Clarified initialization of Status.RP and WatchLo.{I,R,W} bits duringCold Reset in Chapters 4 and 5.

Table of Contents

:

List Of Figures

List Of Tables

:

Introduction to the MIPS32 4K™ Processor Core Family

The MIPS32 4K™ processor cores from MIPS® Technologies are high-performance, low-power, 32-bit MIPS RISC cores intended for custom system-on-silicon applications. The cores are designed for semiconductor manufacturing companies, ASIC developers, and system OEMs who want to rapidly integrate their own custom logic and peripherals with a high-performance RISC processor. The cores are fully synthesizable to allow maximum flexibility; they are highly portable across processes and can be easily integrated into full system-on-silicon designs, allowing developers to focus their attention on end-user products.

The cores are ideally positioned to support new products for emerging segments of the digital consumer, network, systems, and information management markets, enabling new tailored solutions for embedded applications.

The 4K family has three members: the 4Kc™, 4Km™, and 4Kp™ cores. The cores incorporate aspects of both the MIPS Technologies R3000® and R4000® processors. The three devices differ mainly in the type of multiply-divide unit (MDU) and the memory management unit (MMU).

- The 4Kc core contains a fully-associative translation lookaside buffer (TLB) and pipelined MDU.

- The 4Kp core contains a block address translation (BAT) mechanism that is smaller and simpler than the TLB implementation in the 4Kc core, along with a non-pipelined MDU.

- The 4Km processor is really a hybrid of the 4Kc and 4Kp cores. It contains a BAT-based MMU (like the 4Kp core) along with a pipelined MDU (like the 4Kc core).

Optional instruction and data caches are fully programmable from 0 - 16 Kbytes in size. In addition, each cache can be organized as direct-mapped, 2-way, 3-way, or 4-way set associative. On a cache miss, loads are blocked only until the first critical word becomes available. The pipeline resumes execution while the remaining words are being written to the cache. Both caches are virtually indexed and physically tagged. Virtual indexing allows the cache to be indexed in the same clock in which the address is generated rather than waiting for the virtual-to-physical address translation in the Translation Lookaside Buffer (TLB).

All cores execute the MIPS32™ instruction set architecture (ISA). The MIPS32 ISA contains all MIPS II instructions as well as special multiply-accumulate, conditional move, prefetch, wait, and zero/one detect instructions. The R4000-style memory management unit of the 4Kc core contains a 3-entry instruction TLB (ITLB), a 3-entry data TLB(DTLB), and a 16 dual-entry joint TLB (JTLB) with variable page sizes. The 4Kp and

4Km processors cores contain a simplified block address translation (BAT) mechanism where the mapping of address spaces is determined through bits in the Configuration register.

The 4Kc and 4Km multiply-divide unit (MDU) supports a maximum issue rate of one 32x16 multiply (MUL), multiply-add (MADD/MADDU), or multiply-subtract (MSUB/MSUBU) operation per clock, or one 32x32 MUL, MADD, or MSUB every other clock. The basic Enhanced JTAG (EJTAG) features provide CPU run control with stop, single stepping and re-start, and with software breakpoints through the SDBBP instruction. In addition, optional instruction and data virtual address hardware breakpoints, and optional connection to an external EJTAG probe through the Test Access Port (TAP), may be included.

This chapter provides an overview of the MIPS32 4K processor cores and consists of the following sections:

- Section 1.1, "Features"

- Section 1.2, "Block Diagram"

- Section 1.3, "Required Logic Blocks"

- Section 1.4, "Optional Logic Blocks"

1.1 Features

- 32-bit Address and Data Paths

- MIPS32 Compatible Instruction Set

    – All MIPSII™ instructions

    – Multiply-add and multiply-subtract instructions (MADD, MADDU, MSUB, MSUBU)

    – Targeted multiply instruction (MUL)

    – Zero and one detect instructions (CLZ, CLO)

    – Wait instruction (WAIT)

    – Conditional move instructions (MOVZ, MOVN)

    – Prefetch instruction (PREF)

- Programmable Cache Sizes

    – Individually configurable instruction and data caches

    – Sizes from 0 - 16-Kbyte

    – Direct mapped, 2-, 3-, or 4-way set associative

- Loads that miss in the cache are blocked only until critical word is available

- Write-through, no write-allocate

- 16-byte cache line size, word sectored

- Virtually indexed, physically tagged

- Cache line locking support

- Non-blocking prefetches

• R4000 Style Privileged Resource Architecture

- Count/compare registers for real-time timer interrupts

- Instruction and data watch registers for software breakpoints

- Separate interrupt exception vector

• Programmable Memory Management Unit (4Kc core only)

- 16 dual-entry R4000 style JTLB with variable page sizes

- 3-entry instruction TLB

- 3-entry data TLB

• Programmable Memory Management Unit (4Kp and 4Km cores only)

- Block address translation (no JTLB, ITLB, or DTLB)

- Address spaces mapped using register bits

• Simple Bus Interface Unit (BIU)

- All I/Os fully registered

- Separate unidirectional 32-bit address and data buses

- Two 16-byte collapsing write buffers

• Multiply-Divide Unit (4Kc and 4Km cores)

- Max issue rate of one 32x16 multiply per clock

- Max issue rate of one 32x32 multiply every other clock

- Early in divide control. Minimum 11, maximum 34 clock latency on divides

• Power Control

- Minimum frequency: 0 MHz

- Power-down mode (triggered by WAIT instruction)

- Support for software-controlled clock divider

- EJTAG Debug Support

  - CPU control with start, stop and single stepping

  - Software breakpoints via the SDBBP instruction

  - Optional hardware breakpoints on virtual addresses; 4 instruction and 2 data breakpoints, 2 instruction and 1 data breakpoint, or no breakpoints

  - Test Access Port (TAP) facilitates high speed download of application code

1.2 Block Diagram

All cores contain both required and optional blocks. Required blocks are the lightly shaded areas of the block diagram and must be implemented to remain MIPS-compliant. Optional blocks can be added to the cores based on the needs of the implementation. The required blocks are as follows:

- Execution Unit

- Multiply-Divide Unit (MDU)

- System Control Coprocessor (CP0)

- Memory Management Unit (MMU)

- Cache Controller

- Bus Interface Unit (BIU)

- Power Management

Optional blocks include:

- Instruction Cache

- Data Cache

- Enhanced JTAG (EJTAG) Controller

Figure 1-1 shows a block diagram of a 4K core. The MMU can be implemented using either a translation lookaside buffer (TLB) in the case of the 4Kc core, or a fixed block address translator (BAT) in the case of the 4Kp and 4Km cores. Refer to Chapter 3 for more information.
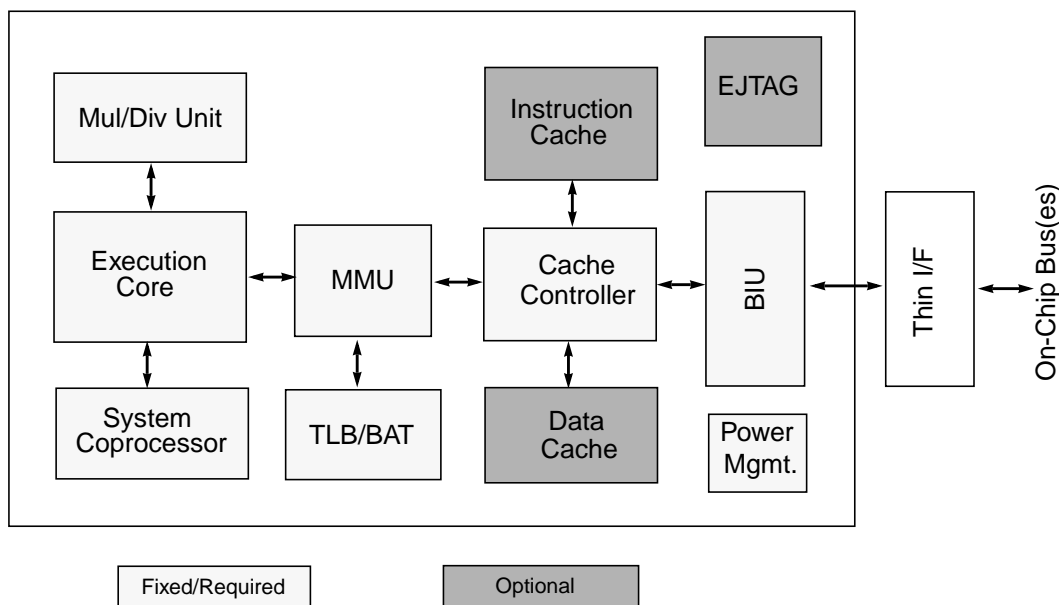
Figure 1-1   4K Processor Core Block Diagram

1.3 Required Logic Blocks

The following subsections describe the various required logic blocks of the 4K processor cores.

1.3.1   Execution Unit

The core execution unit implements a load-store architecture with single-cycle Arithmetic Logic Unit (ALU) operations (logical, shift, add, subtract) and an autonomous multiply-divide unit. The core contains thirty-two 32-bit general-purpose registers used for scalar integer operations and address calculation. The register file consists of two read ports and one write port and is fully bypassed to minimize operation latency in the pipeline.

The execution unit includes:

- 32-bit adder used for calculating the data address

- Address unit for calculating the next instruction address

- Logic for branch determination and branch target address calculation

- Load aligner

- Bypass multiplexers used to avoid stalls when executing instruction streams where data- producing instructions are followed closely by consumers of their results

- Zero/One detect unit for implementing the CLZ and CLO instructions

- Arithmetic Logic Unit (ALU) for performing bitwise logical operations

- Shifter and Store Aligner

1.3.2 Multiply/Divide Unit (MDU)

The Multiply/Divide unit performs multiply and divide operations. In the 4Kc and 4Km processors, the MDU consists of a 32x16 booth-encoded multiplier, result-accumulation registers (HI and LO), a divide state machine, and all multiplexers and control logic required to perform these functions. This pipelined MDU supports execution of a 16x16 or 32x16 multiply operation every clock cycle; 32x32 multiply operations can be issued every other clock cycle. Appropriate interlocks are implemented to stall the issue of back-to-back 32x32 multiply operations. Divide operations are implemented with a simple 1 bit per clock iterative algorithm and require 35 clock cycles in worst case to complete. Early-in to the algorithm detects sign extension of the dividend, if it is actual size is 24, 16 or 8 bit. the divider will skip 7, 15 or 23 of the 32 iterations. An attempt to issue a subsequent MDU instruction while a divide is still active causes a pipeline stall until the divide operation is completed.

In the 4Kp processor, the non-pipelined MDU consists of a 32-bit full-adder, result-accumulation registers (HI and LO), a combined multiply/divide state machine, and all multiplexers and control logic required to perform these functions. It performs any multiply using 32 cycles in an iterative 1 bit per clock algorithm. Divide operations are also implemented with a simple 1 bit per clock iterative algorithm (no early-in) and require 35 clock cycles to complete. An attempt to issue a subsequent MDU instruction while a multiply/divide is still active causes a pipeline stall until the operation is completed.

All cores implement an additional multiply instruction, MUL, which specifies that lower 32-bits of the multiply result be placed in the primary register file instead of the HI/LO register pair. By avoiding the explicit Move From LO (MFLO) instruction, required when using the LO register, and by supporting multiple destination registers, the throughput of multiply-intensive operations is increased.

Two instructions, multiply-add (MADD/MADDU) and multiply-subtract (MSUB/MSUBU), are used to perform the multiply-add and multiply-subtract operations. The MADD instruction multiplies two numbers and then adds the product to the current contents of the HI and LO registers. Similarly, the MSUB instruction multiplies two operands and then subtracts the product from the HI and LO registers. The MADD/MADDU and MSUB/MSUBU operations are commonly used in Digital Signal Processor (DSP) algorithms.

1.3.3   System Control Coprocessor (CP0)

In the MIPS architecture, CP0 is responsible for the virtual-to-physical address translation and cache protocols, the exception control system, the processor's diagnostics capability, operating mode selection (kernel vs. user mode), and the enabling and disabling of interrupts. Configuration information such as cache size, set associativity, and EJTAG debug features are available by accessing the CP0 registers. Refer to Chapter 6 for more information on the CP0 registers. Refer to Chapter 9 for more information on EJTAG debug registers.

1.3.4   Memory Management Unit (MMU)

Each core contains an MMU that interfaces between the execution unit and the cache controller. Although the 4Kc core implements a 32-bit architecture, the Memory Management Unit (MMU) is modeled after the MMU found in the 64-bit R4000 family.

The 4Kc core implements a translation lookaside buffer (TLB). The TLB consists of three translation buffers: a 16 dual-entry fully associative Joint TLB (JTLB), a 3-entry fully associative Instruction TLB (ITLB) and a 3-entry fully associative data TLB(DTLB). The ITLB and DTLB (the micro TLBs) are managed by the hardware and are not software visible. The micro TLBs contain subsets of the JTLB. When translating addresses, the corresponding micro TLB (I or D) is accessed first. If there is not a matching entry, the JTLB is used to translate the address and refill the micro TLB. If the entry is not found in the JTLB, an exception is taken. To minimize the micro TLB miss penalty, the JTLB is looked up in parallel with the DTLB for data references. This results in a 1 cycle stall for a DTLB miss and a 2 cycle stall for an ITLB miss.

The 4Kp and 4Km cores implement a block address translation (BAT) mechanism instead of a TLB. The BAT replaces both the JTLB and ITLB in the 4Kc core. The BAT performs a simple translation to get the physical address from the virtual address. Refer to Chapter 3 for more information on the BAT.

Figure 1-2 shows how the ITLB/BAT, DTLB/BAT, JTLB are used.
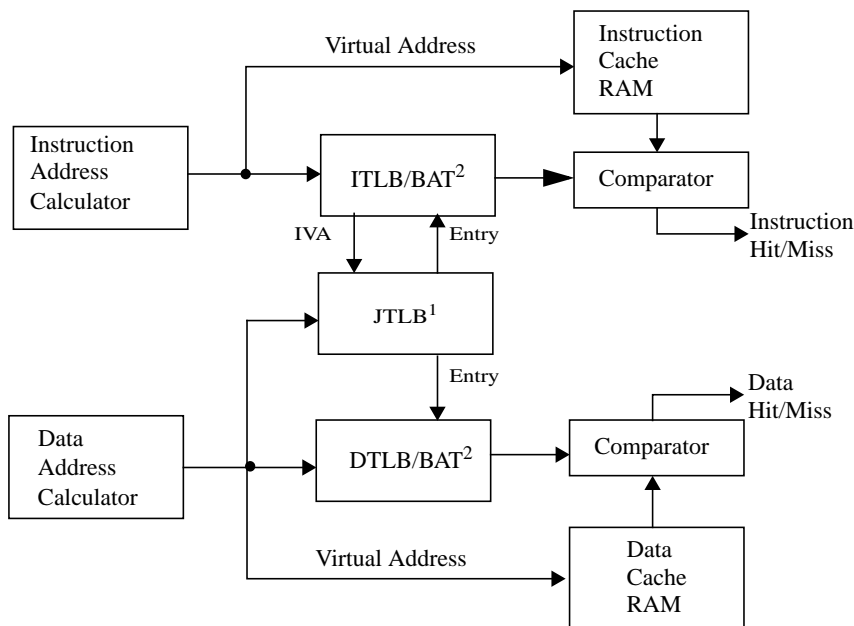
1. JTLB only exists in the 4Kc core.
2. ITLB/DTLB implemented in the 4Kc core only. BAT implemented in the 4Kp and 4Km cores.

Figure 1-2   Address Translation During a Cache Access

1.3.5   Cache Controllers

The data and instruction cache controllers support caches of various sizes, organizations, and set associativity. For example, the data cache can be 2 Kbytes in size and 2-way set associative, while the instruction cache can be 8 Kbytes in size and 4-way set associative.

Each cache controller contains and manages a one line fill buffer. Besides accumulating data to be written to the cache, the fill buffer is accessed in parallel with the cache and data can be bypassed back to the core.

Refer to Chapter 7 for more information on the instruction and data cache controllers.

1.3.6   Bus Interface Unit(BIU)

The Bus Interface Unit (BIU) controls the external interface signals. Additionally, it contains the implementation of the 32-byte collapsing write buffer. The purpose of this buffer is to hold and combine write transactions before issuing them at the external interface. Since the data caches for all cores follow a write-through cache policy, the write buffer significantly reduces the number of write transactions on the external interface as well as reducing the amount of stalling in the core due to issuance of multiple writes in a short period of time.

The write buffer is organized as two 16-byte buffers. Each buffer contains data from a single 16-byte aligned block of memory. One buffer contains the data currently being transferred on the external interface, while the other buffer contains accumulating data from the core.

Refer to Chapter 6 for more information on the BIU.

1.3.7   Power Management

The core offers a number of power management features, including low-power design, active power management, and power-down modes of operation. The core is a static design that supports a WAIT instruction designed to signal the rest of the device that execution and clocking should be halted, reducing system power consumption during idle periods.

The core provides two mechanisms for system-level, low-power support:

*   Register-controlled power management

*   Instruction-controlled power management

In register controlled power management mode the core provides three bits in the CP0 Status register for software control of the power management function and allows interrupts to be serviced even when the core is in power-down mode. In instruction controlled power-down mode execution of the WAIT instruction is used to invoke low-power mode.

Refer to Chapter 8 for more information on power management.

1.4 Optional Logic Blocks

The core consists of the following optional logic blocks as shown in the block diagram in Figure 1-1.

1.4.1   Instruction Cache

The instruction cache is an optional on-chip memory array of up to 16 Kbytes. The cache is virtually indexed and physically tagged, allowing the virtual-to-physical address translation to occur in parallel with the cache access rather than having to wait for the physical address translation. The tag holds 22 bits of the physical address, 4 valid bits, a lock bit, and the FIFO replacement bit.

All cores support instruction cache-locking. Cache locking allows critical code to be locked into the cache on a "per-line" basis, enabling the system designer to maximize the efficiency of the system cache. Cache locking is always available on all instruction cache entries. Entries can be marked as locked or unlocked on a per-entry basis using the CACHE instruction.

1.4.2   Data Cache

The data cache is an optional on-chip memory array of up to 16-Kbytes. The cache is virtually indexed and physically tagged, allowing the virtual-to-physical address translation to occur in parallel with the cache access. The tag holds 22 bits of the physical address, 4 valid bits, a lock bit, and the FIFO replacement bit.

In addition to instruction cache locking, all cores also support a data cache locking mechanism identical to the instruction cache, with critical data segments to be locked into the cache on a "per-line" basis. The locked contents cannot be selected for replacement on a cache miss, but can be updated on a store hit.

Cache locking is always available on all data cache entries. Entries can be marked as locked or unlocked on a per-entry basis using the CACHE instruction.

1.4.3   EJTAG Controller

All cores provide basic EJTAG support with debug mode, run control, single step and software breakpoint instruction (SDBBP) as part of the core. These features allow for the basic software debug of user and kernel code.

Optional EJTAG features include hardware breakpoints. A 4K core may have four instruction breakpoints and two data breakpoints, two instruction breakpoints and one data breakpoint, or no breakpoints. The hardware instruction breakpoints can be configured to generate a debug exception when an instruction is executed anywhere in the virtual address space. Bit mask and address space identifier (ASID) values may apply in the address compare. These breakpoints are not limited to code in RAM like the software instruction breakpoint (SDBBP). The data breakpoints can be configured to generate a debug exception on a data transaction. The data transaction may be qualified with both virtual address, data value, size and load/store transaction type. Bit mask and ASID values may apply in the address compare, and byte mask may apply in the value compare.

Refer to the Chapter 9 for more information on hardware breakpoints.

 An optional Test Access Port (TAP) provides for the communication from an EJTAG probe to the CPU through a dedicated port, may also be applied to the core. This provides the possibility for debugging without debug code in the application, and for download of application code to the system.

Refer to Chapter 6 for a list of EJTAG interface signals. Refer to Chapter 9 for more information on the EJTAG features.

Pipeline

The MIPS32 4K™ processor cores implement a 5-stage pipeline similar to the original R3000 pipeline. The pipeline allows the processor to achieve high frequency while minimizing device complexity, reducing both cost and power consumption. This chapter contains the following sections:

- Section 2.1, "Pipeline Stages"

- Section 2.2, "Instruction Cache Miss"

- Section 2.3, "Data Cache Miss"

- Section 2.4, "Multiply/Divide Operations"

- Section 2.5, "MDU Pipeline (4Kc and 4Km Cores)"

- Section 2.6, "MDU Pipeline (4Kp Core Only)"

- Section 2.7, "Branch Delay"

- Section 2.8, "Interlock Handling"

- Section 2.9, "Slip Conditions"

- Section 2.10, "Instruction Interlocks"

2.1 Pipeline Stages

The pipeline consists of five stages:

- Instruction (I Stage)

- Execution (E Stage)

- Memory (M Stage)

- Align/Accumulate (A Stage)

- Writeback (W stage)

All three cores implement a bypass mechanism that allows the result of an operation to be forwarded directly to the instruction that needs it without having to write the result to the register and then read it back.

Figure 2-1 shows the operations performed in each pipeline stage of the 4Kc processor.
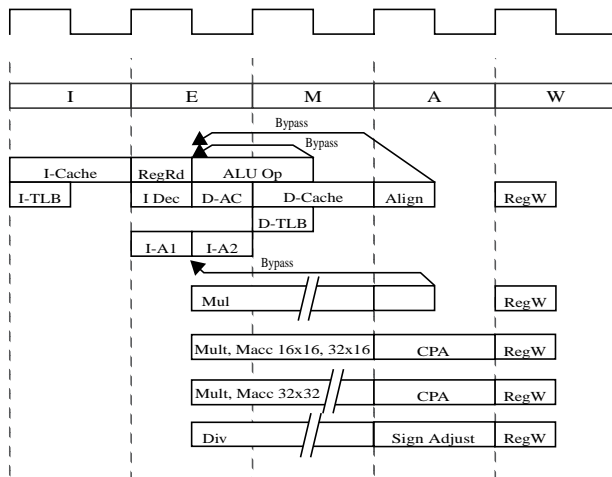
Figure 2-1   4Kc Core Pipeline Stages

Figure 2-2 shows the operations performed in each pipeline stage of the 4Km processor core.
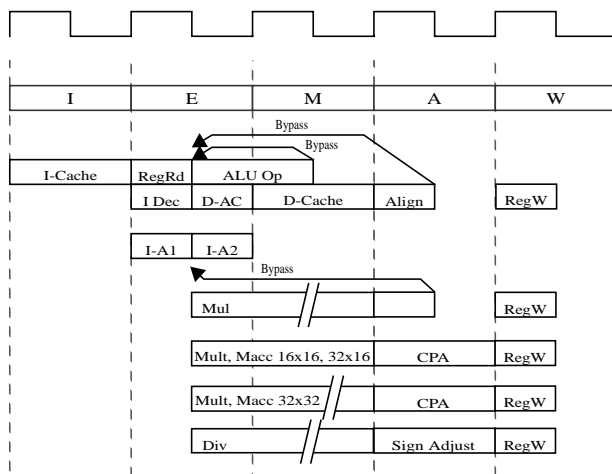
Figure 2-2   4Km Core Pipeline Stages

Figure 2-3 shows the operations performed in each pipeline stage of the 4Kp processor core.
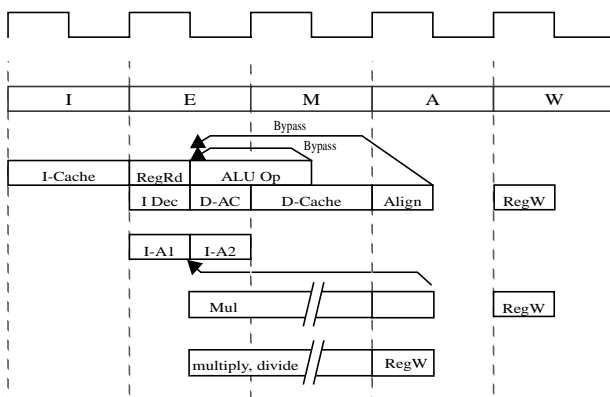
Figure 2-3   4Kp Core Pipeline Stages

2.1.1   I Stage: Instruction Fetch

During the Instruction fetch stage:

- The instruction translation lookaside buffer (I-TLB) perform a virtual-to-physical address translation (4Kc core only).

- An instruction is fetched from the instruction cache.

2.1.2   E Stage: Execution

During the Execution stage:

- Operands are fetched from the register file.

- The Arithmetic Logic Unit (ALU) begins the arithmetic or logical operation for register-to-register instructions.

- The ALU calculates the data virtual address for load and store instructions.

- The ALU determines whether the branch condition is true and calculates the virtual branch target address for branch instructions.

- Instruction logic selects an instruction address

- All multiply and divide operations begin in this stage.

2.1.3   M Stage: Memory Fetch

During the Memory Fetch stage:

- The arithmetic or logic ALU operation completes.

- The data cache fetch and the data virtual-to-physical address translation are performed for load and store instructions.

- Data TLB (4Kc core only) and data cache lookup are performed and a hit/miss determination is made.

- A 16x16 or 32x16 MUL operation completes in the array and stalls for one clock in the M stage to complete the carry-propagate-add in the M stage (4Kc and 4Km cores).

- A 32x32 MUL operation stalls for two clocks in the M stage to complete second cycle of the array and the carry-propagate-add in the M stage (4Kc and 4Km cores).

- A 16x16 or 32x16 MULT/MADD/MSUB operation completes in the array (4Kc and 4Km cores).

- A 32x32 MULT/MADD/MSUB operation stalls for one clock in the M_{MDU} stage of the MDU pipeline to complete second cycle in the array (4Kc and 4Km cores).

- A divide operation stalls for a maximum of 32 clocks in the M_{MDU} stage of the MDU pipeline (4Kc and 4Km cores).

- A multiply operation stalls for 31 clocks in M_{MDU} stage (4Kp core only).

- A multiply-accumulate operation stalls for 33 clocks in M_{MDU} stage (4Kp core only).

- A divide operation stalls for 32 clocks in the M_{MDU} stage (4Kp core only).

2.1.4   A Stage: Align/Accumulate

During the Align/Accumulate stage:

- A separate aligner aligns load data with its word boundary.

- A MULT/MADD/MSUB operation performs the carry-propagate-add. This includes the accumulate step for the MADD/MSUB operations. The actual register writeback is performed in the W stage (4Kc and 4Km cores).

- A MUL operation makes the result available for writeback. The actual register writeback is performed in the W stage (all 4K cores).

- A divide operation perform the final Sign-Adjust. The actual register writeback is performed in the W stage (4Kc and 4Km cores)

- A multiply/divide operation writes to HI/LO registers (4Kp core only).

2.1.5   W Stage: Writeback

- For register-to-register or load instructions, the result is written back to the register file during the W stage.

2.2 Instruction Cache Miss

When the instruction cache is indexed, the instruction address is translated to determine if the required instruction resides in the cache. An instruction cache miss occurs when the requested instruction address does not reside in the instruction cache. When a cache miss is detected in the I stage, the core transitions to the E stage. The pipeline stalls in the E stage until the miss is resolved. The bus interface unit must select the address from multiple sources. If the address bus is busy, the request will remain in this arbitration stage (B-ASel in Figure 2-4) until the bus is available. The core drives the selected address onto the bus. The number of clocks required to access the bus is determined by the access time of the array that contains the data. The number of clocks required to return the data once the bus is accessed is also determined by the access time of the array.

Once the data is returned to the core, the critical word is written to the instruction register for immediate use. The bypass mechanism allows the core to use the data once it becomes available, as opposed to having the entire cache line written to the instruction cache, then reading out the required word.

Figure 2-4 shows a timing diagram of an instruction cache miss.
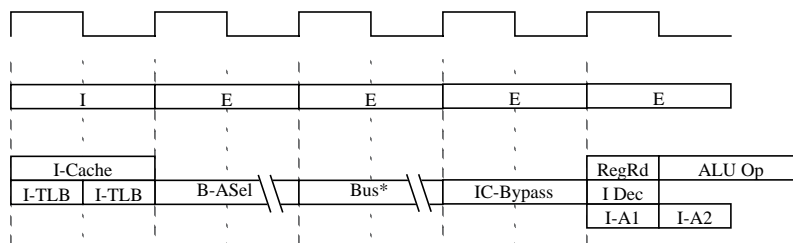
* Contains all of the time that address and data are utilizing the bus.

Figure 2-4   Instruction Cache Miss Timing

2.3 Data Cache Miss

When an instruction is indexed, the instruction address is translated to determine if the required instruction resides in the cache. A data cache miss occurs when the requested data address does not reside in the data cache.

When a data cache miss is detected in the M stage (D-TLB), the core transitions to the A stage. The pipeline stalls in the A stage until the miss is resolved (requested data is returned). The bus interface unit arbitrates between multiple requests and selects the correct address to be driven onto the bus (B-ASel in Figure 2-5). The core drives the selected address onto the bus. The number of clocks required to access the bus is determined by the access time of the array containing the data. The number of clocks required to return the data once the bus is accessed is also determined by the access time of the array.

Once the data is returned to the core, the critical word of data passes through the aligner before being forwarded to the execution unit and register file. The bypass mechanism allows the core to use the data once it becomes available, as opposed to having the entire cache line written to the data cache, then reading out the required word.

Figure 2-5 shows a timing diagram of a data cache miss.

* Contains all of the time that address and data are utilizing the bus.
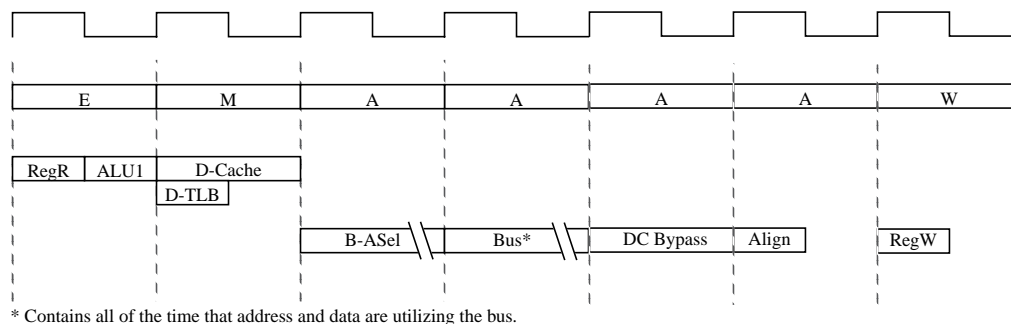
Figure 2-5   Load/Store Cache Miss Timing

2.4 Multiply/Divide Operations

All three cores implement the standard MIPS II™ multiply and divide instructions. Additionally, several new instructions were added for enhanced code performance.

The targeted multiply instruction, MUL, specifies that multiply results be placed in the general purpose register file instead of the HI/LO register pair. By avoiding the explicit MFLO instruction, required when using the LO register, and by supporting multiple destination registers, the throughput of multiply-intensive operations is increased.

Four instructions, multiply-add (MADD), multiply-add-unsigned (MADDU) multiply-subtract (MSUB), and multiply-subtract-unsigned (MSUBU), are used to perform the multiply-accumulate and multiply-subtract operations. The MADD/MADDU instruction multiplies two numbers and then adds the product to the current

contents of the HI and LO registers. Similarly, the MSUB/MSUBU instruction multiplies two operands and then subtracts the product from the HI and LO registers. The MADD/MADDU and MSUB/MSUBU operations are commonly used in DSP algorithms.

All multiply operations (except the MUL instruction) write to the HI/LO register pair. All integer operations write to the general purpose registers (GPR). Because MDU operations write to different registers than integer operations, following integer instructions can execute before the MDU operation has completed. The MFLO and MFHI instructions are used to move data from the HI/LO register pair to the GPR file. If a MFLO or MFHI instruction is issued before the MDU operation completes, it will stall to wait for the data.

2.5 MDU Pipeline (4Kc and 4Km Cores)

The 4Kc and 4Km processor cores contain a multiply/divide unit (MDU) with a separate pipeline for multiply and divide operations. This pipeline operates in parallel with the integer unit (IU) pipeline and does not stall when the IU pipeline stalls. This allows long-running MDU operations, such as a divide, to be partially masked by system stalls and/or other integer unit instructions.

The MDU consists of a 32x16 booth encoded multiplier, result/accumulation registers (HI and LO), a divide state machine, and all necessary multiplexers and control logic. The first number shown ('32' of 32x16) represents the *rs* operand. The second number ('16' of 32x16) represents the *rt* operand. The core only checks the latter (*rt*) operand value to determine how many times the operation must pass through the multiplier. The 16x16 and 32x16 operations pass through the multiplier once. A 32x32 operation passes through the multiplier twice.

The MDU supports execution of a 16x16 or 32x16 multiply operation every clock cycle; 32x32 multiply operations can be issued every other clock cycle. Appropriate interlocks are implemented to stall the issue of back-to-back 32x32 multiply operations. Multiply operand size is automatically determined by logic built into the MDU. Divide operations are implemented with a simple 1 bit per clock iterative algorithm with an early in detection of sign extension on the dividend (*rs*). Any attempt to issue a subsequent MDU instruction while a divide is still active causes an IU pipeline stall until the divide operation is completed.

Table 2-1 lists the latencies (number of cycles until a result is available) for multiply and divide instructions. The latencies are listed in terms of pipeline clocks. In this table 'latency' refers to the number of cycles necessary for the first instruction to produce the result needed by the second instruction.

Table 2-1   4Kc and 4Km Core Instruction Latencies

Size of operand 1st Instruction[1]	Instruction Sequence		Latency
	1st Instruction	2nd instruction	
16 bit	MULT/MULTU, MADD/MADDU, or MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU, or MFHI/MFLO	1
32 bit	MULT/MULTU, MADD/MADDU, or MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU, or MFHI/MFLO	2
16 bit	MUL	Integer operation[2]	2[3]
32 bit	MUL	Integer operation[2]	2[3]
8 bit	DIVU	MFHI/MFLO	12
16 bit	DIVU	MFHI/MFLO	19
24 bit	DIVU	MFHI/MFLO	26
32 bit	DIVU	MFHI/MFLO	33
8 bit	DIV	MFHI/MFLO	13[4]
16 bit	DIV	MFHI/MFLO	20[4]
24 bit	DIV	MFHI/MFLO	27[4]
32 bit	DIV	MFHI/MFLO	34[4]
any	MFHI/MFLO	Integer operation[2]	2
any	MTHI/MTLO	MADD/MADDU, or MSUB/MSUBU	1

[1] For multiply operations this is the *rt* operand. For divide operations this is the *rs* operand.

[2] Integer Operation refers to any integer instruction that uses the result of a previous MDU operation.

[3] This does not include the 1 or 2 IU pipeline stalls (16 bit or 32 bit) that MUL operation causes irrespective of the following instruction.

[4] If both operands are positive the Sign Adjust stage is bypassed. Timing is then the same as for DIVU.

In Table 2-1 a latency of one means that the first and second instruction can be issued back to back in the code without the MDU causing any stalls in the IU pipeline. A latency of two means that if issued back to back, the IU pipeline will be stalled for one cycle. MUL operations are special because it needs to stall the IU pipeline in order to maintain its register file write slot. Consequently the MUL 16x16 or 32x16 operation will always force a one cycle stall of the IU pipeline, and the MUL 32x32 will force a two cycle stall. If the integer instruction immediately following the MUL operation uses its result, an additional stall is forced on the IU pipeline.

Table 2-2 lists the repeat rates (peak issue rate of cycles until the operation can be reissued) for multiply accumulate/subtract instructions. The repeat rates are listed in terms of pipeline clocks. In this table 'repeat rate' refers to the case where the first MDU instruction (in the table below) if back to back with the second instruction.

Table 2-2   4Kc Core Instruction Repeat Rates

Operand Size of 1st Instruction	Instruction Sequence		Repeat Rate
	1st Instruction	2nd instruction	
16 bit	MULT/MULTU, MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU	1
32 bit	MULT/MULTU, MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU	2

Table 2-3 below shows the pipeline flow for the following sequence:

      1.   32x16 multiply (M_1)

2.   Add

3.   32x32 multiply (M_2)

The 32x16 multiply operation requires one clock of each pipeline stage to complete. The 32x32 requires two clocks in the M_{MDU} stage. The MDU pipeline is shown as the shaded areas of Table 2-3 and always starts a computation in the final phase of the E stage. As shown in the table, the M_{MDU} stage of the MDU pipeline occurs in parallel with the M stage of the IU pipeline, the A_{MDU} stage occurs in parallel with the A stage, and the W_{MDU} stage occurs in parallel with the W stage.

Table 2-3   MDU Pipeline Behavior During Multiply Operations (4Kc & 4Km Processors)

Clock	I	E	M M_{MDU}	A A_{MDU}	W W_{MDU}
1	M_1				
2	ADD	M_1			
3	M_2	ADD			
			M_1		
4		M_2	ADD		
				M_1	
5				ADD	
			M_2		M_1
6					ADD
			M_2		
7					
				M_2	
8					
					M_2

The following is a clock-by-clock analysis of Table 2-3.

1. The first 32x16 multiply operation (M_1) enters the I stage and is fetched from the instruction cache.

2. An ADD operation enters the I stage. The M_1 operation enters the E stage. The integer and MDU pipelines share the I and E pipeline stages. At the end of the E stage in clock 2, the multiply operation (M_1) is passed to the MDU pipeline.

3. In clock 3 a 32x32 multiply operation (M_2) enters the I stage and is fetched from the instruction cache. Since the ADD operation has not yet reached the M stage by clock 3, there is no activity in the M stage of the integer pipeline at this time.

4. In clock 4 the second multiply operation (M_2) enters the E stage. The ADD operation enters M stage of the integer pipe. Since the M_1 multiply is a 32x16 operation, only one clock is required for the M_{MDU} stage, hence the M_1 operation passes to the A_{MDU} stage of the MDU pipeline.

5. In clock 5 the M_2 multiply enters the M_{MDU} stage. The ADD operation enters the A stage of the integer pipeline. The M_1 operation completes and is written back in to the HI/LO register pair in the W_{MDU} stage.

6. Since a 32x32 multiply requires two passes through the multiplier, with each pass requiring one clock, the 32x32 remains in the M_{MDU} stage in clock 6. The ADD operation completes and is written to the register file in the W stage of the integer pipeline.

7. The M_2 multiply operation progresses to the A_{MDU} stage

8. The M_2 operation completes and is written to the HI/LO registers pair the W_{MDU} stage.

2.5.1 32x16 Multiply (4Kc & 4Km Cores)

The 32x16 multiply operation begins in the last phase of the E stage, which is shared between the integer and MDU pipelines. In the latter phase of the E stage, the *rs* and *rt* operands arrive and the booth recoding function occurs at this time. The multiply calculation requires one clock and occurs in the M_{MDU} stage. In the A_{MDU} stage, the carry-propagate-add function occurs and the operation is completed. The result is written back to the HI/LO register pair in the first half of the W_{MDU} stage.

Figure 2-6 shows a diagram of a 32x16 multiply operation.
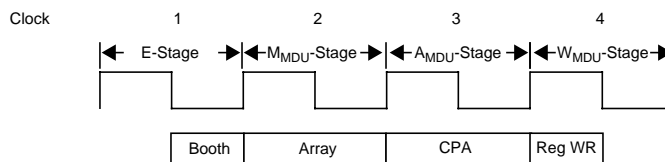
Figure 2-6   MDU Pipeline Flow During a 32x16 Multiply Operation

2.5.2 32x32 Multiply (4Kc & 4Km Cores)

The 32x32 multiply operation begins in the last phase of the E stage, which is shared between the integer and MDU pipelines. In the latter phase or the E stage, the *rs* and *rt* operands arrive and the booth recoding function

occurs at this time. The multiply calculation requires two clocks and occurs in the M_{MDU} stage. In the A_{MDU} stage, the carry-propagate-add (CPA) function occurs and the operation is completed. The result is written back to the HI/LO register pair in the first half of the W_{MDU} stage.

Figure 2-7 shows a diagram of a 32x32 multiply operation.
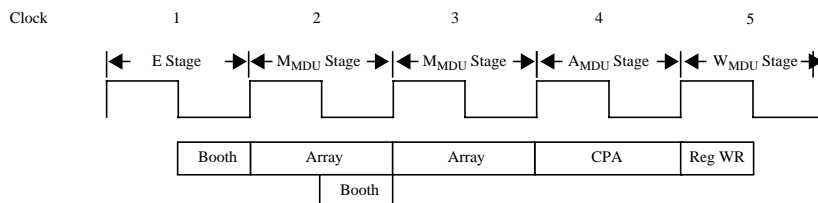
Figure 2-7   MDU Pipeline Flow During a 32x32 Multiply Operation

2.5.3   Divide (4Kc & 4Km Cores)

Divide operations are implemented using a simple non-restoring division algorithm. This algorithm works only for positive operands, hence the first cycle of the M_{MDU} stage is used to negate the *rs* operand (RS Adjust) if needed. Note that this cycle is executed even if the adjustment is not necessary. At maximum the next 32 clocks (3-34) execute an iterative add/subtract function. In cycle 3 an early in detection is performed in parallel with the add/subtract. The adjusted *rs* operand is detected to be zero extended on the upper most 8, 16 or 24 bits. If this is the case the following 7, 15 or 23 cycles of the add/subtract iterations are skipped.

The remainder adjust (Rem Adjust) cycle is required if the remainder was negative. Note that this cycle is taken even if the remainder was positive. A sign adjust is performed on the quotient and/or remainder if necessary. Note that the sign adjust cycle is skipped if both operands are positive. In this case the Rem Adjust is moved to the A_{MDU} stage.

Figure 2-8, Figure 2-9, Figure 2-10 and Figure 2-11 shows the latency for a 8, 16, 24 and 32 bit divide operation. The repeat rate is either 12, 20, 28 or 35 cycles (one less if the *sign adjust* stage is skipped) as a second divide can be in the *RS Adjust* stage when the first divide is in the *Reg WR* stage.
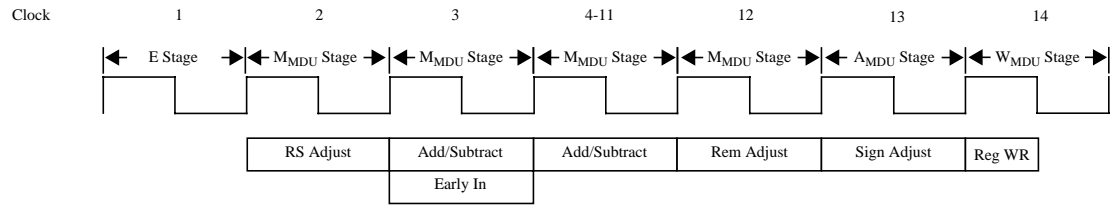
Figure 2-8   MDU Pipeline Flow During a 8 bit Divide Operation
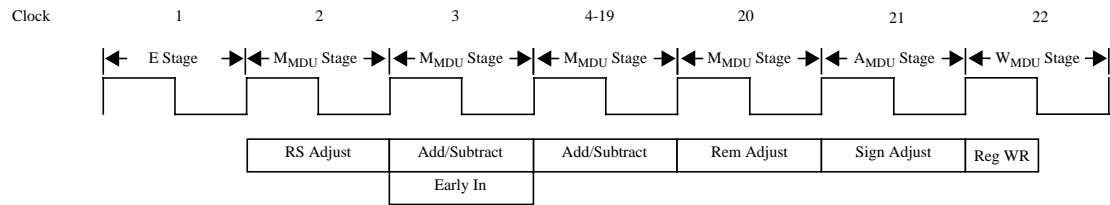
Figure 2-9   MDU Pipeline Flow During a 16 bit Divide Operation
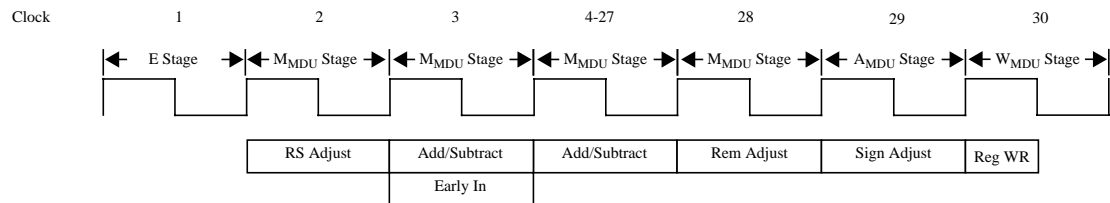
Figure 2-10 MDU Pipeline Flow During a 24 bit Divide Operation
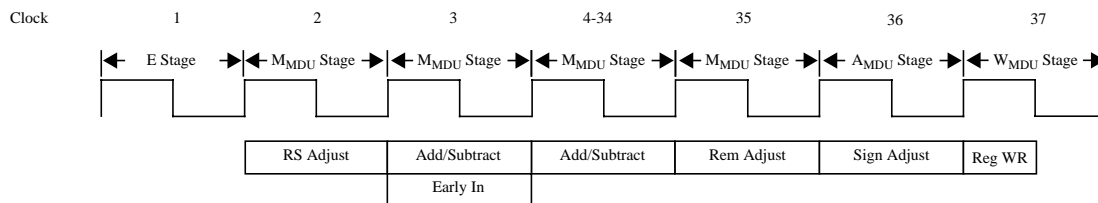
Figure 2-11 MDU Pipeline Flow During a 32 bit Divide Operation

2.6 MDU Pipeline (4Kp Core Only)

The multiply/divide unit (MDU) is a separate pipeline for multiply and divide operations. This pipeline operates in parallel with the integer unit (IU) pipeline and does not stall when the IU pipeline stalls. This allows the long-running MDU operations to be partially masked by system stalls and/or other integer unit instructions.

The MDU consists of one 32-bit adder result-accumulate registers (HI and LO), a combined multiply/divide state machine and all multiplexers and control logic. A simple 1-bit per clock recursive algorithm is used for both multiply and divide operations. Using booth's algorithm all multiply operations complete in 32 clocks. Two extra clocks are needed for multiply-accumulate. The non-restoring algorithm used for divide operations will not work with negative numbers. Adjustment before and after are thus required depending on the sign of the operands. All divide operations complete in 33 to 35 clocks.

Table 2-4 lists the latencies (number of cycles until a result is available) for multiply and divide instructions. The latencies are listed in terms of pipeline clocks. In this table 'latency' refers to the number of cycles necessary for the second instruction to use the results of the first.

Table 2-4   4Kp Core Instruction Latencies

Operand Signs of 1st Instruction (Rs,Rt)	Instruction Sequence		Latency
	1st Instruction	**2nd instruction**	
any, any	MULT/MULTU	MADD/MADDU, MSUB/MSUBU, or MFHI/MFLO	32

Table 2-4   4Kp Core Instruction Latencies

Operand Signs of 1st Instruction (Rs,Rt)	Instruction Sequence		Latency
	1st Instruction	**2nd instruction**	
any, any	MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU, or MFHI/MFLO	34
any, any	MUL	Integer operation[1]	32
any, any	DIVU	MFHI/MFLO	33
pos, pos	DIV	MFHI/MFLO	33
any, neg	DIV	MFHI/MFLO	34
neg, pos	DIV	MFHI/MFLO	35
any, any	MFHI/MFLO	Integer operation[1]	2
any, any	MTHI/MTLO	MADD/MADDU, MSUB/MSUBU	1
[1] Integer Operation refers to any integer instruction that uses the result of a previous MDU operation.			

2.6.1   Multiply (4Kp Core)

Multiply operations implement a simple iterative multiply algorithm. Using Booth's approach, this algorithm works for both positive and negative operands. The operation uses 32 cycles in M_{MDU} stage to complete a multiplication. The register writeback to HI and LO are done in the A stage. For MUL operations, the register file writeback is done in the W_{MDU} stage.

Figure 2-12 shows the latency for a multiply operation. The repeat rate is 33 cycles as a second multiply can be in the first M_{MDU} stage when the first multiply is in A_{MDU} stage.
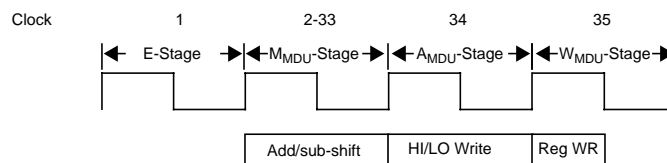
Figure 2-12   4Kp MDU Pipeline Flow During a Multiply Operation

2.6.2 Multiply Accumulate (4Kp Core)

Multiply-accumulate operations use the same multiply machine as used for multiply only. Two extra stages are needed to perform the addition/subtraction. The operations uses 34 cycles in M_{MDU} stage to complete the multiply-accumulate. The register writeback to HI and LO are done in the A stage.

Figure 2-13 shows the latency for a multiply-accumulate operation. The repeat rate is 35 cycles as a second multiply-accumulate can be in the E stage when the first multiply is in the last M_{MDU} stage.
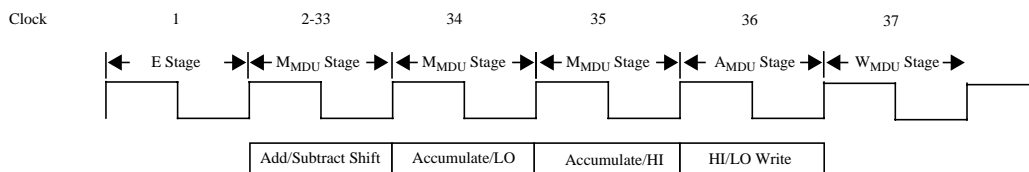
Figure 2-13 4Kp MDU Pipeline Flow During a Multiply Accumulate Operation

2.6.3 Divide (4Kp Core)

Divide operations also implement a simple non-restoring algorithm. This algorithm works only for positive operands, hence the first cycle of the M_{MDU} stage is used to negate the rs operand (RS Adjust) if needed. Note that this cycle is executed even if negation is not needed. The next 32 cycle (3-34) executes an interactive add/subtract-shift function.

Two sign adjust (Sign Adjust 1/2) cycles are used to change the sign of one or both the quotient and the remainder. Note that one or both of these cycles are skipped if they are not needed. The rule is, if both operands were positive or if this is an unsigned division; both of the sign adjust cycles are skipped. If the *rs* operand was negative, one of the sign adjust cycles is skipped. If only the *rs* operand was negative, none of the sign adjust cycles are skipped. Register writeback to HI and LO are done in the A stage.

Figure 2-11 shows the latency for a divide operation. The repeat rate is either 34, 35 or 36 cycles (depending on how many sign adjust cycles are skipped) as a second divide can be in the E stage when the first divide is in the last M_{MDU} stage.
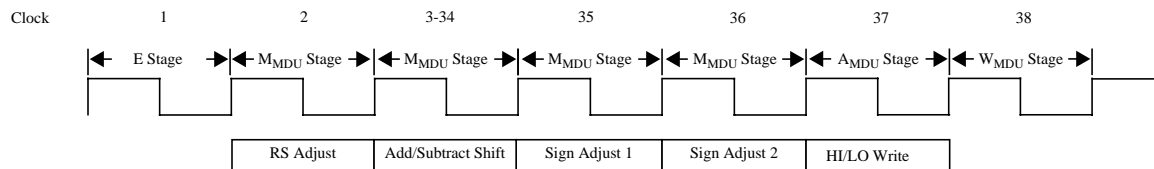
Figure 2-14 4Kp MDU Pipeline Flow During a Divide Operation

2.7 Branch Delay

The pipeline has a branch delay of one cycle and a load delay of one cycle. The one-cycle branch delay is a result of the branch decision logic operating during the E pipeline stage. This allows the branch target address calculated in the previous stage to be used for the instruction access in the following E stage. The branch delay slot means that no bubbles are injected into the pipeline on branch instructions. The address calculation and branch condition check are both performed in the E stage. The target PC is used for the next instruction in the I stage (2nd instruction after the branch).

The pipeline begins the fetch of either the branch path or the fall-through path in the cycle following the delay slot. After the branch decision is made, the processor continues with the fetch of either the branch path (for a taken branch) or the fall-through path (for the non-taken branch).

Figure 2-15 illustrates the branch delay.
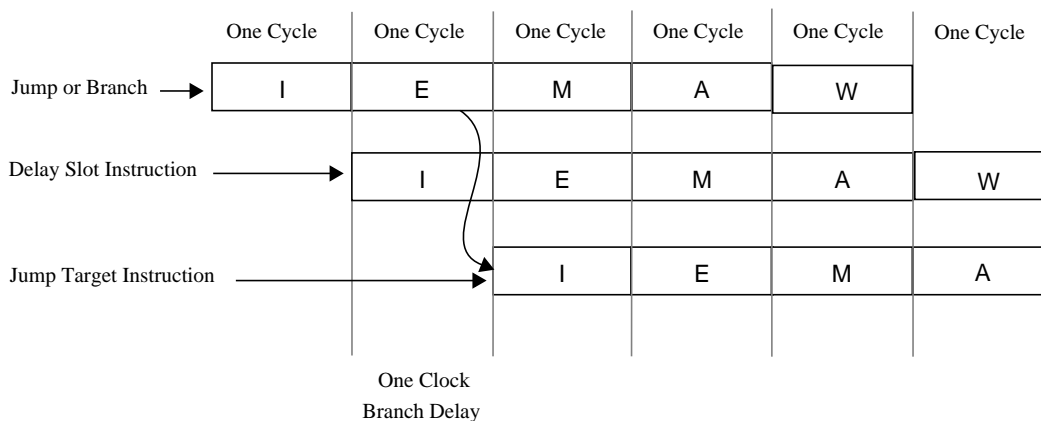
Figure 2-15 CPU Pipeline Branch Delay

2.8 Interlock Handling

Smooth pipeline flow is interrupted when cache misses occur or when data dependencies are detected. Interruptions handled using hardware, such as cache misses, are referred to as *interlocks*. At each cycle, interlock conditions are checked for all active instructions.

Table 2-5 lists the types of pipeline interlocks for the 4K processor cores.

Table 2-5   Pipeline Interlocks

Interlock Type	Sources	Slip Stage
ITLB Miss	Instruction TLB	I Stage
ICache Miss	Instruction cache	E Stage
Instruction	Producer-consumer hazards	E/M Stage
	Hardware Dependencies (MDU/TLB)	E Stage
DTLB Miss	Data TLB	M Stage

Table 2-5   Pipeline Interlocks

Interlock Type	Sources	Slip Stage
Data Cache Miss	Load that misses in data cache	W Stage
	Multi-cycle cache Op	
	Sync	
	Store when write thru buffer full	
	EJTAG breakpoint on store	
	VA match needing data value comparison	
	Store hitting in fill buffer	

In general, MIPS processors support two types of hardware interlocks:

- Stalls, which are resolved by halting the pipeline

- Slips, which allow one part of the pipeline to advance while another part of the pipeline is held static

In the 4K processor cores, all interlocks are handled as slips.

2.9 Slip Conditions

On every clock internal logic determines whether each pipe stage is allowed to advance. These slip conditions propagate backwards down the pipe. For example, if the M stage does not advance, neither will the E or I stages.

Slipped instructions are retried on subsequent cycles until they issue. The back end of the pipeline advances normally during slips in an attempt to resolve the conflict. NOPS are inserted into the bubble in the pipeline. Figure 2-16 shows an instruction cache miss.

Clock   1   2   3   4   5   6

① Cache miss detected

② Critical word received

③ Execute E-stage
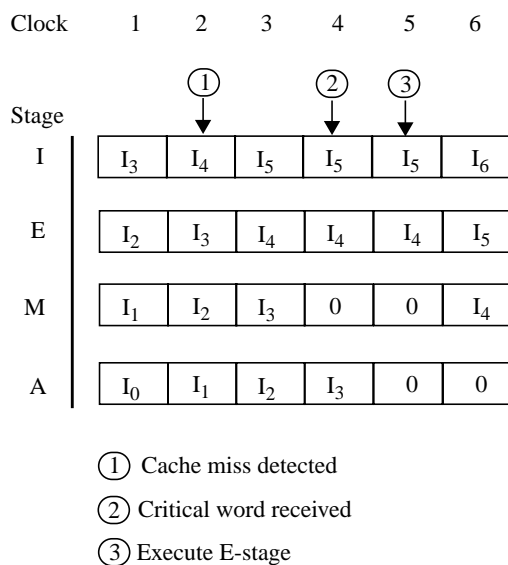
Figure 2-16 Instruction Cache Miss Slip

Figure 2-16 shows a diagram of a two-cycle slip. In the first clock cycle, the pipeline is full and the cache miss is detected. Instruction I0 is in the A stage, instruction I1 is in the M stage, instruction I2 is in the E stage, and instruction I3 is in the I stage. The cache miss occurs in clock 2 when the I4 instruction fetch is attempted. I4 advances to the E-stage and waits for the instruction to be fetched from main memory. In this example it takes two clocks (3 and 4) to fetch the I4 instruction from memory. Once the cache miss is resolved in clock 4 and the instruction is bypassed to the cache, the pipeline is restarted, causing the I4 instruction to finally execute it's E-stage operations.

2.10 Instruction Interlocks

Most instructions can be issued at a rate of one per clock cycle. In some cases, in order to ensure a sequential programming model, the issue of an instruction is delayed to ensure that the results of a prior instruction will be available. Table 2-6 details the instruction interactions that delay the issuance of an instruction into the processor pipeline.

Table 2-6   Instruction Interlocks

Instruction Interlocks				
First Instruction		**Second Instruction**	**Issue Delay (in Clock Cycles)**	**Slip Stage**
LB/LBU/LH/LHU/LL/LW/LWL/LWR		Consumer of load data	1	E stage
MFC0		Consumer of destination register	1	E stage
MULT/MADD/MSUB (4Kc and 4Km cores)	16bx32b	MFLO/MFHI	0	M stage
	32bx32b		1	M stage
MUL (4Kc and 4Km cores)	16bx32b	Consumer of target data	2	E stage
	32bx32b		3	E stage
MUL (4Kc and 4Km cores)	16bx32b	Non-Consumer of target data	1	E stage
	32bx32b		2	E stage
MFHI/MFLO		Consumer of target data	1	E stage
MULT/MADD/MSUB (4Kc and 4Km cores)	16bx32b	MULT/MUL/MADD/MSUB MTHI/MTLO/DIV	0	E stage
	32bx32b		1	E stage
DIV		MULT/MUL/MADD/MSUB /MTHI/MTLO/MFHI/MFLO/DIV	Until DIV completes	E stage
MULT/MUL/MADD/MSUB/MTHI/MTLO/ MFHI/MFLO/DIV (4Kp core)		MULT/MUL/MADD/MSUB /MTHI/MTLO/MFHI/MFLO/DIV	Until 1st MDU op completes	E stage
MUL (4Kp core)		Any Instruction	Until MUL completes	E stage
MFC0		Consumer of target data	1	E stage
TLBWR/TLBWI		Load/Store/PREF/CACHE/Cop0 op	2	E stage
TLBR			1	E stage

Memory Management

The MIPS32 4K™ processor cores contain a Memory Management Unit (MMU) that interfaces between the execution unit and the cache controller. The 4Kc core implements a Translation Lookaside Buffer (TLB), while the 4Kp and 4Km cores implement a simpler block address translation (BAT) scheme.

This chapter contains the following sections:

- Section 3.1, "Translation Lookaside Buffer (4Kc Core Only)"

- Section 3.2, "TLB Instructions (4Kc Core)"

- Section 3.3, "Block Address Translation (4Kp & 4Km Cores)"

- Section 3.4, "Modes of Operation"

- Section 3.5, "System Control Coprocessor"

In the 4Kc processor core, the TLB consists of three address translation buffers: a 16 dual-entry fully associative Joint TLB (JTLB), a 3-entry Instruction micro TLB (ITLB), and a 3-entry Data micro TLB (DTLB). When an address is translated, the appropriate micro TLB (ITLB or DTLB) is accessed first. If the translation is not found in the micro TLB, the JTLB is accessed. If there is a miss in the JTLB, an exception is taken.

In the 4Kp and 4Km processor cores, the BAT translates virtual addresses into physical addresses via a fixed translation mechanism. These translations are different for the different regions of the virtual address space (USeg/KUSeg, KSeg0, KSeg1, KSeg2/3).

In the 4Kp and 4Km cores, note that the BAT replaces the ITLB and DTLB found in the 4Kc core, and that the JTLB is not used.

Figure 3-1 shows how the ITLB, DTLB, JTLB, and BAT are implemented.
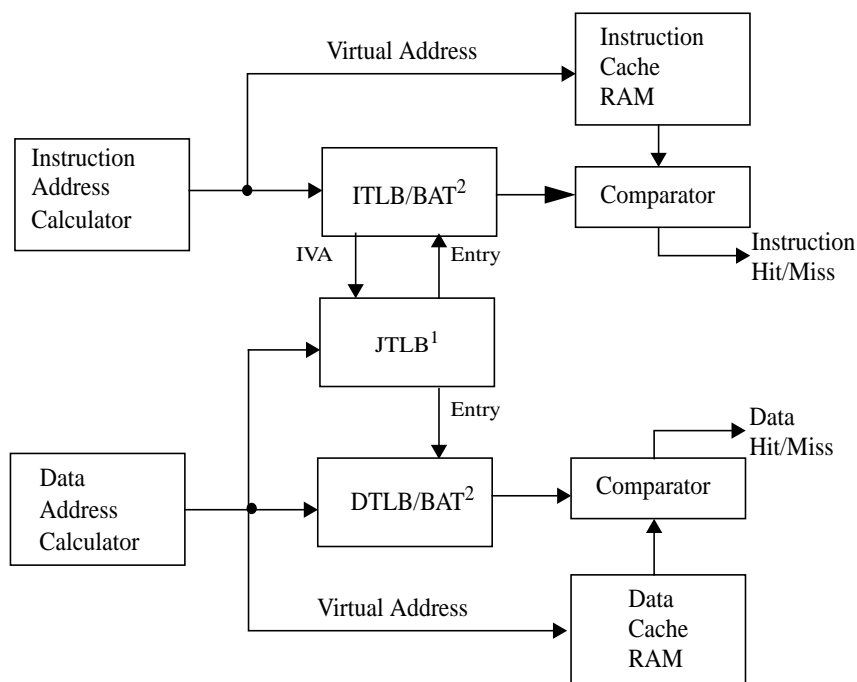
1. JTLB only implemented in the 4Kc core.
2. ITLB/DTLB implemented in the 4Kc core only. BAT implemented in the 4Kp and 4Km cores.

Figure 3-1   Address Translation During a Cache Access

3.1 Translation Lookaside Buffer (4Kc Core Only)

The following subsections discuss the TLB memory management scheme used in 4Kc processor core. The TLB consists of two address translation buffers:

- 16 dual-entry fully associative Joint TLB (JTLB)

- 3-entry fully associative Instruction TLB (ITLB)

- 3-entry fully associative Data TLB (DTLB)

3.1.1   Joint TLB (4Kc Core)

The 4Kc core implements a 16 dual-entry, fully associative JTLB that maps 32 virtual pages to their corresponding physical addresses. The JTLB is organized as 16 pairs of even and odd entries containing pages that range in size from 4-Kbytes to 16-Mbytes into the 4-Gbyte physical address space. The purpose of the TLB is to translate virtual addresses and their corresponding ASID into a physical memory address. The translation is performed by comparing the upper bits of the virtual address (along with the address space identifier(ASID)) against each of the entries in the *tag* portion of the joint TLB structure.

The JTLB is organized in page pairs to minimize the overall size. Each *tag* entry corresponds to 2-data entries, an even page entry and an odd page entry. The highest order virtual address bit not participating in the tag comparison is used to determine which of the data entries is used. Since page size can vary on a page-pair basis, the determination of which address bits participate in the comparison and which bit is used to make the even-odd determination must be determined dynamically during the TLB lookup.

3.1.2   Instruction TLB (4Kc Core)

The ITLB is a small 3-entry, fully associative TLB dedicated to performing translations for the instruction stream. The ITLB only maps 4-Kbyte pages/sub-pages.

The ITLB is managed by hardware and is transparent to software. If a fetch address cannot be translated by the ITLB, the JTLB is used to attempt to translate it in the following clock cycle. If successful, the translation information is copied into the ITLB. The ITLB is then re-accessed and the address will be successfully translated. This results in an ITLB miss penalty of at least 2 cycles (If the JTLB is busy with other operations, it may take additional cycles)

3.1.3 Data TLB (4Kc Core)

The DTLB is a small 3-entry, fully associative TLB which provides a faster translation for Load/Store addresses than is possible with the JTLB. The DTLB only maps 4-Kbyte pages/sub-pages.

Like the ITLB, the DTLB is managed by hardware and is transparent to software. Unlike the ITLB, when translating Load/Store addresses, the JTLB is accessed in parallel with the DTLB. If there is a DTLB miss and a JTLB hit, the DTLB can be reloaded that cycle. The DTLB is then re-accessed and the translation will be successful. This parallel access reduces the DTLB miss penalty to 1 cycle.

3.1.4 Virtual to Physical Address Translation (4Kc Core)

Converting a virtual address to a physical address begins by comparing the virtual address from the processor with the virtual addresses in the TLB. There is a match when the virtual page number (VPN) of the address is the same as the VPN field of the entry, and either:

- The Global (G) bit of both the even and odd pages of the TLB entry are set, or

- The ASID field of the virtual address is the same as the ASID field of the TLB entry

This match is referred to as a TLB *hit*. If there is no match, a TLB *miss* exception is taken by the processor and software is allowed to refill the TLB from a page table of virtual/physical addresses in memory.

Figure 3-2 shows the logical translation of a virtual address into a physical address.

In this figure the virtual address is extended with an 8-bit address-space identifier (ASID), which reduces the frequency of TLB flushing during a context switch. This 8-bit ASID contains the number assigned to that process and is stored in the CP0 *EntryHi* register.

1.Virtual address (VA) represented by the virtual page number (VPN) is compared with tag in TLB.

2. If there is a match, the page frame number (PFN) representing the upper bits of the physical address (PA) is output from the TLB.

3. The Offset, which does not pass through the TLB, is then concatenated with the PFN.

Virtual address

G	ASID	VPN		Offset

TLB

G	ASID	VPN

	PFN

TLB Entry

PFN		Offset

Physical address

Figure 3-2   Overview of a Virtual-to-Physical Address Translation in the 4Kc Core

If there is a virtual address match in the TLB, the physical address is output from the TLB and concatenated with the *Offset*, which represents an address within the page frame space. The *offset* does not pass through the TLB.

Figure 3-3 shows a flow diagram of the 4Kc core address translation process. The top portion of the figure shows a virtual address for a 4-Kbyte page size. The width of the *offset* is defined by the page size. The remaining 20 bits of the address represent the virtual page number (VPN), and index the 1M-entry page table.

The bottom portion of Figure 3-3 shows the virtual address for a 16-Mbyte page size. The remaining 8 bits of the address represent the VPN, and index the 256-entry page table.

Virtual address with 1M (2^{20}) 4-Kbyte pages
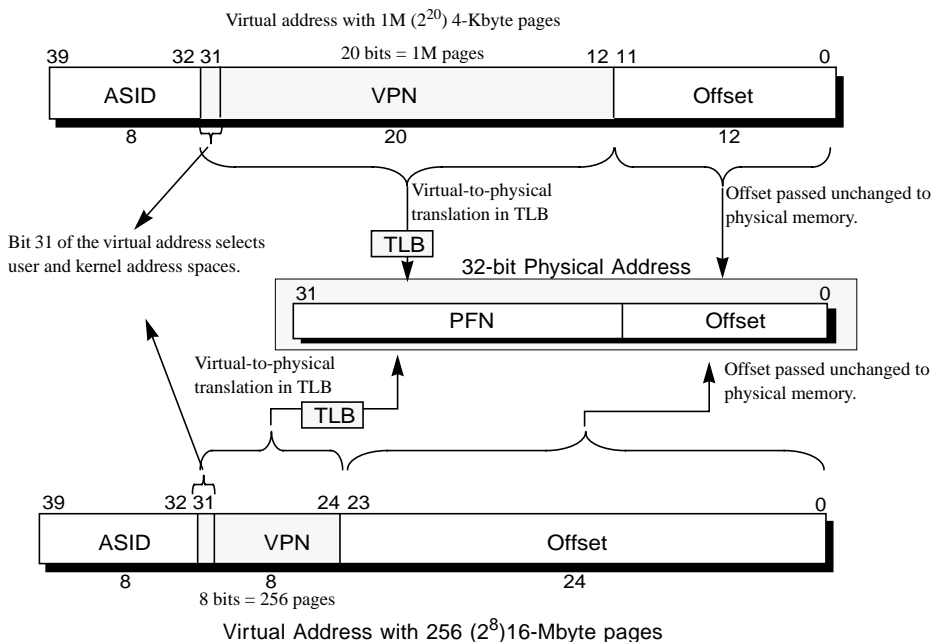
Figure 3-3   32-bit Virtual Address Translation

3.1.5   Hits, Misses, and Multiple Matches (4Kc Core)

Each JTLB entry contains a tag portion and a data portion. If a match is found, the upper bits of the virtual address are replaced with the page frame number (PFN) stored in the corresponding entry in the data array of the joint TLB (JTLB). The granularity of JTLB mappings is defined in terms of TLB *pages*. The 4Kc core JTLB supports pages of different sizes ranging from 4-Kbyte to 16-MB in powers of 4. If a match is found, but the entry is invalid, a TLB Invalid exception is taken.

If no match occurs (TLB miss), an exception is taken and software refills the TLB from the page table resident in memory. Software can write over a selected TLB entry or use a hardware mechanism to write into a random entry. In addition, there is a hidden bit in each TLB entry that is cleared on a ColdReset. This bit is set once the TLB entry is written and is included in the match detection. Therefore, uninitialized TLB entries will not cause a TLB shutdown.

The 4Kc core implements a TLB write-compare mechanism to ensure that multiple TLB matches do not occur. On the TLB write operation, the write value is compared with all other entries in the TLB. If a match occurs, the

4Kc core takes a machine-check exception, sets the TS bit in the CP0 *Status* register, and aborts the write operation.

Note: To be compatible with other MIPS processors, it is recommended that software initialize all TLB entries with unique tag values and V bits cleared before the first access to a memory mapped location.

Table 3-1 shows the address bits used for even/odd bank selection depending on page size and the relationship between the legal values in the mask register and the selected page size.

Table 3-1   Mask and Page Size Values

PageMask[11:0]	Page Size	Even/Odd Bank Select Bit
0000_0000_0000	4KB	VAddr[12]
0000_0000_0011	16KB	VAddr[14]
0000_0000_1111	64KB	VAddr[16]
0000_0011_1111	256KB	VAddr[18]
0000_1111_1111	1MB	VAddr[20]
0011_1111_1111	4MB	VAddr[22]
1111_1111_1111	16MB	VAddr[24]

3.1.6   Page Sizes and Replacement Algorithm (4Kc Core)

To assist in controlling both the amount of mapped space and the replacement characteristics of various memory regions, the 4Kc core provides two mechanisms. First, the page size can be configured, on a per entry basis, to map a page size of 4 kbyte to 16 Mbyte (in multiples of 4). The CP0 PageMask register is loaded with the mapping page size, which is then entered into the TLB when a new entry is written. Thus, operating systems can provide special-purpose maps. For example, a typical frame buffer can be memory mapped with only one TLB entry.

The second mechanism controls the replacement algorithm when a TLB miss occurs. To select a TLB entry to be written with a new mapping, the 4Kc core provides a random replacement algorithm. However, the processor also provides a mechanism whereby a programmable number of mappings can be locked into the TLB via the Wired register, thus avoiding random replacement.

3.1.7 TLB Tag and Data Formats (4Kc Core)

Figure 3-4 shows the format of a TLB *tag* entry. The entry is divided into the following fields:

- Global process indicator (G bit)

- Address space identifier

- Virtual page number

- Compressed page mask

Setting the G bit indicates that the entry is global to all processes and/or threads in the system. In this case, the 8-bit ASID value is ignored since the entry is not relative to a specific thread or process.

The address space identifier (ASID) helps to reduce the frequency of TLB flushing on a context switch. The existence of the ASID allows multiple processes to exist in both the TLB and instruction caches. The ASID value is stored in the EntryHi register and is compared to the ASID value of each entry. Figure 3-4 and Table 3-2 show the TLB tag entry format. Figure 3-5 and Table 3-3 show the TLB data array entry format.

G	ASID[7:0]	VPN2[31:25]	VPN2[24:13]	CMASK[5:0]
1	8	7	12	6
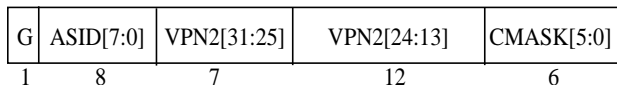
Figure 3-4   TLB Tag Entry Format

Table 3-2   TLB Tag Entry Fields

Field Name	Description
G	Global Bit. When set, indicates that this entry is global to all processes and/or threads and thus disables inclusion of the ASID in the comparison.
ASID[7:0]	Address Space Identifier. Identifies which process or thread this TLB entry is associated with.
VPN2[31:25], VPN2[24:13]	Virtual Page Number divided by 2. This field contains the upper bits of the virtual page number. Because it represents a pair of TLB pages, it is divided by 2. Bits 31:25 are always included in the TLB lookup comparison. Bits 24:13 are included depending on the page size.
CMASK[5:0]	Compressed Page Mask Value. This field is a compressed version of the page mask. It defines the page size by masking the appropriate VPN2 bits from being involved in a comparison. It is also used to determine which address bit is used to make the even-odd page determination.

PFN[31:12]	C[2:0]	D	V
20	3	1	1

Figure 3-5   TLB Data Array Entry Format

Table 3-3   TLB Data Array Entry Fields

Field Name	Description
PFN[31:12]	Physical Frame Number. Defines the upper bits of the physical address. For page sizes larger than 4 Kbytes, only a subset of these bits is actually used.
C[2:0]	Cacheability. Contains an encoded value of the cacheability attributes and determines whether the page should be placed in the cache or not. The field is encoded as follows: | CS[2:0]	Coherency Attribute
:--	:--
000	Maps to entry 011b*
001	Maps to entry 011b*
010	Uncached
011	Cacheable, noncoherent, write-through, no write allocated
100	Maps to entry 011b*
101	Maps to entry 011b*
110	Maps to entry 011b*
111	Maps to entry 010b* | * These mappings are not used on the 4K processor cores but do have meaning in other MIPS Technologies implementations. Refer to the MIPS32 specification for more information.
D	"Dirty" or Write-enable Bit. Indicates that the page has been written, and/or is writable. If this bit is set, stores to the page are permitted. If the bit is cleared, stores to the page cause a TLB Modified exception.
V	Valid Bit. Indicates that the TLB entry and, thus, the virtual page mapping are valid. If this bit is set, accesses to the page are permitted. If the bit is cleared, accesses to the page cause a TLB Invalid exception.

3.2 TLB Instructions (4Kc Core)

Table 3-4 lists the 4Kc core TLB-related instructions. Refer to Chapter 11 for more information on these instructions.

Table 3-4   TLB Instructions

Op Code	Description of Instruction
TLBP	Translation Lookaside Buffer Probe
TLBR	Translation Lookaside Buffer Read
TLBWI	Translation Lookaside Buffer Write Index
TLBWR	Translation Lookaside Buffer Write Random
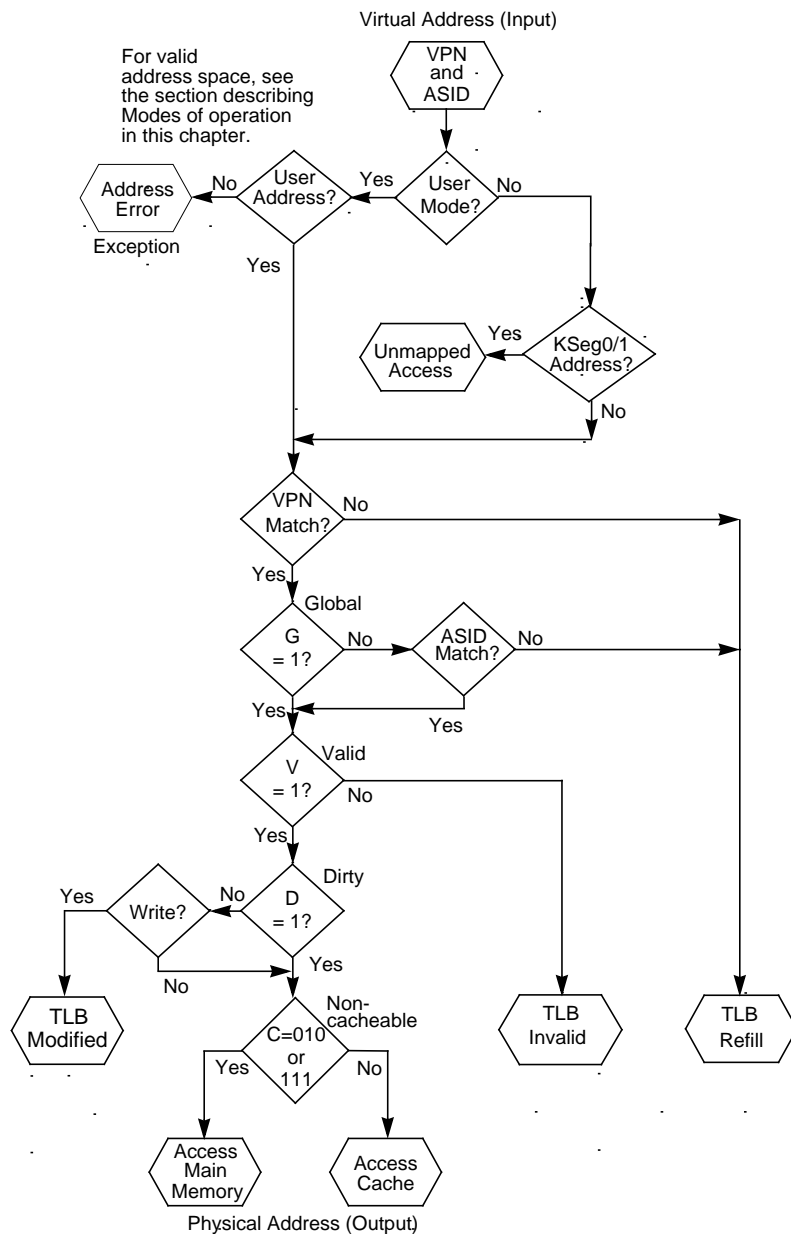
Figure 3-6   TLB Address Translation Flow in the 4Kc Processor Core

3.3 Block Address Translation (4Kp & 4Km Cores)

The 4Kp and 4Km cores implement a simple block address translation (BAT) mechanism that is smaller than the 4Kc TLB and more easily synthesized. Like the 4Kc TLB, the BAT performs virtual-to-physical address translation and provides attributes for the different segments. Those segments which are unmapped in the 4Kc TLB implementation (kseg0 and kseg1) are translated identically by the BAT.

The BAT also determines the cacheability of each segment. These attributes are controlled via bits in the Confide register. Table 3-5 shows the encoding for the K23 (bits 30:28), KU (bits 27:25) and K0 (bits 2:0) of the Config register.

Table 3-5   Cache Coherency Attributes

Config Register Fields K23, KU, and K0	Cache Coherency Attribute
0, 1, 3, 4, 5, 6	Cacheable, noncoherent, write through, no write allocate
2, 7	Uncached

In the 4Kp & 4Km cores, no translation exceptions can be taken, although address errors are still possible.

Table 3-6   Cacheability of Segments with Block Address Translation

Segment	Virtual Address Range	Cacheability
USeg/KUSeg	0x0000_0000-0x7FFF_FFFF	Controlled by the KU field (bits 27:25) of the Config register. Refer to Table 3.5 for the encoding.
KSeg0	0x8000_0000-0x9FFF_FFFF	Controlled by the K0 field (bits 2:0) of the Config register. See Table 3-5 for the encoding.
KSeg1	0xA000_0000-0xBFFF_FFFF	Always uncacheable
KSeg2	0xC000_0000-0xDFFF_FFFF	Controlled by the K23 field (bits 30:28) of the Config register. Refer to Table 3.5 for the encoding.
KSeg3	0xE000_0000-0xFFFF_FFFF	Controlled by K23 field (bits 30:28) of the Config register. Refer to Table 3.5 for the encoding.

The BAT performs a simple translation to map from virtual addresses to physical addresses. This mapping is shown in Figure 3-7. When ERL=1, USeg and KUSeg become unmapped and uncached. This behavior is the same as if there was a JTLB. This mapping is shown in Figure 3-8.
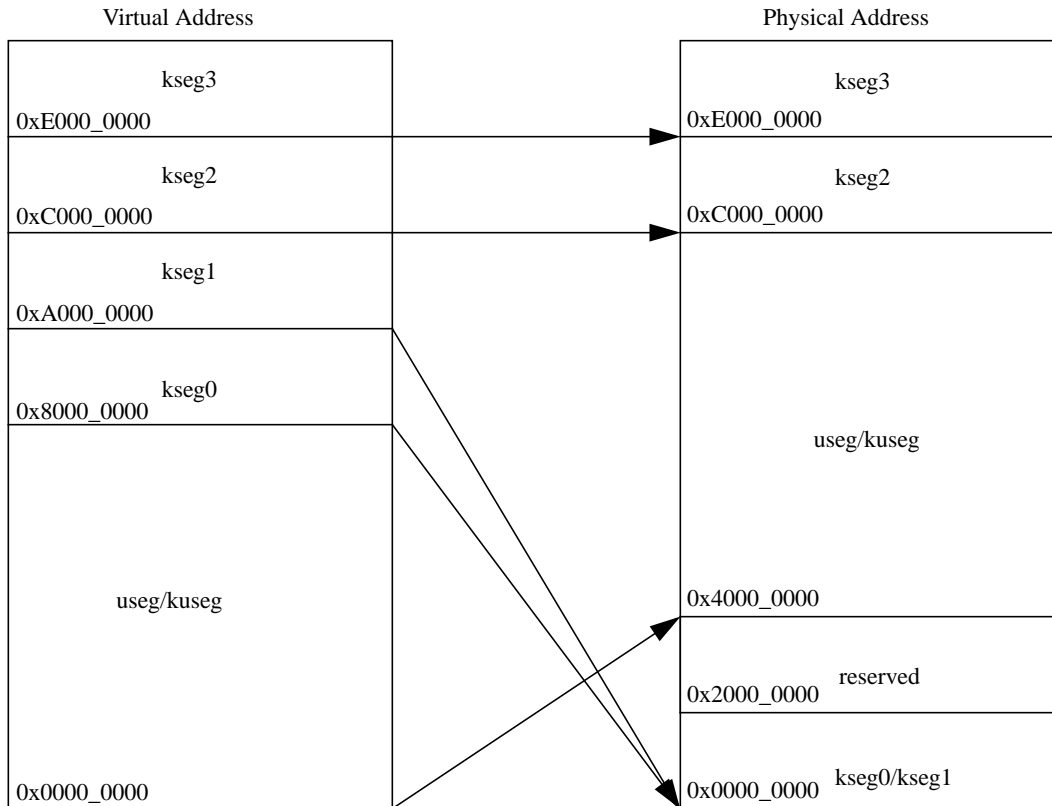
Figure 3-7   BAT Memory Map (ERL=0) in the 4Kp and 4Km Processor Cores
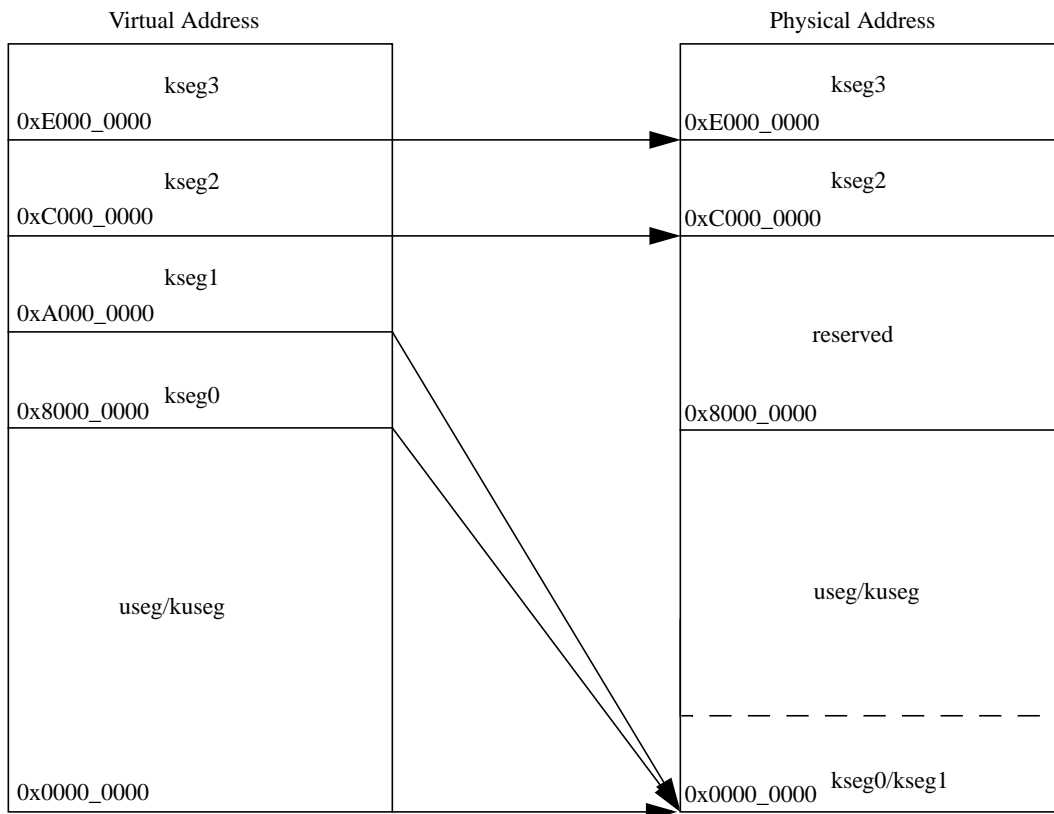
Figure 3-8   BAT Memory Map (ERL=1) in the 4Kp and 4Km Processor Cores

3.4 Modes of Operation

All 4K processor cores support three modes of operation: user mode, kernel mode and debug mode. User mode is most often used for applications programs. kernel mode is typically used for handling exceptions and operating system kernel functions, including CP0 management and I/O device accesses.

The core enters kernel mode both at reset and when an exception is recognized. While in kernel mode, software has access to the entire address space, as well as all CP0 registers. User mode accesses are limited to a subset of the virtual address space (0x0000_0000 to 0x7FFF_FFFF) and can be inhibited from accessing CP0 functions. In user mode, addresses 0x8000_0000 to 0xFFFF_FFFF are invalid and cause an exception if accessed.

Debug mode is entered on a debug exception. While in debug mode the debug software has access to the same address space and CP0 registers as for kernel mode, and in addition, access to the debug area in the address space.

3.4.1   User Mode

In user mode, a single 2 G-byte (2^{31} bytes) uniform virtual address space called user segment (useg) is available. Figure 3-9 shows the location of user mode virtual address space.

32-bit

```
0xFFFF_FFFF ┌──────────┐
            │ Unmapped │
            │(Address  │
            │  Error)  │
0x8000_0000 ├──────────┤
0x7FFF_FFFF │          │
            │  2 GB    │
            │  Mapped  │  useg
            │          │
0x0000_0000 └──────────┘
```
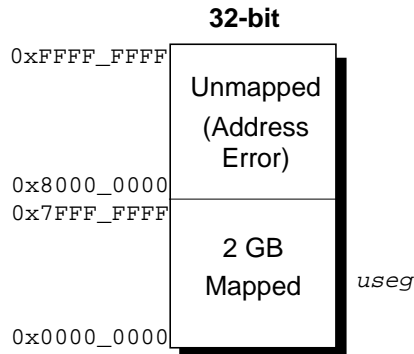
Figure 3-9   User Mode Virtual Address Space

The user segment starts at address 0x0000_0000 and ends at address 0x7FFF_FFFF. Accesses to all other addresses cause an address error exception.

The processor operates in user mode when the *Status* register contains the following bit values:

* $UM = 1$

- *EXL* = 0

- *ERL* = 0

In addition to the above values, the DM bit in the *Debug* register must be 0.

Table 3-7 lists the characteristics of the *useg* user mode segments.

Table 3-7   User Mode Segments

Address Bit Value	Status Register Bit Value			Segment Name	Address Range	Segment Size
	EXL	ERL	UM			
32-bit A(31) = 0	0	0	1	useg	0x0000_0000 --> 0x7FFF_FFFF	2 Gbyte (2^{31} bytes)

All valid user mode virtual addresses have their most-significant bit cleared to 0, indicating that user mode can only access the lower half of the virtual memory map. Any attempt to reference an address with the most-significant bit set while in user mode causes an address error exception.

The system maps all references to *useg* through the TLB, and bit settings within the TLB entry for the page determine the cacheability of a reference.

3.4.2   Kernel Mode

The processor operates in kernel mode when the DM bit in the *Debug* register is 0 and the *Status* register contains one or more of the following values:

- *UM* = 0

- *ERL*= 1

- *EXL* = 1

When a non-debug exception is detected, EXL or ERL will be set and the processor will enter kernel mode. At the end of exception handler routine, an Exception Return (ERET) instruction is generally executed. The ERET instruction jumps to the Exception PC, clears ERL, and clears EXL if ERL=0. This may return the processor to user mode.

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 3-10 which lists the characteristics of the kernel mode segments.

```
0xFFFF_FFFF ┌──────────────────────────┐ ┐
            │  Kernel virtual address space │ │
            │           Mapped          │ │
0xFF40_0000 ├──────────────────────────┤ │
0xFF3F_FFFF │  Kernel virtual address space │ ├ kseg3
0xFF20_0000 ├──────────────────────────┤ │
0xF1FF_FFFF │  Kernel virtual address space │ │
            │           Mapped          │ │
0xE000_0000 ├──────────────────────────┤ ┘
0xDFFF_FFFF │  Kernel virtual address space │
            │      Mapped, 512 MB      │   kseg2
0xC000_0000 ├──────────────────────────┤
0xBFFF_FFFF │  Kernel virtual address space │
            │     Unmapped, 512 MB     │   kseg1
            │          Uncached         │
0xA000_0000 ├──────────────────────────┤
0x9FFF_FFFF │  Kernel virtual address space │
            │     Unmapped, 512 MB     │   kseg0
0x8000_0000 ├──────────────────────────┤
0x7FFF_FFFF │                          │
            │                          │
            │                          │
            │                          │
            │                          │
            │      Mapped, 2048 MB     │   kuseg
            │                          │
            │                          │
            │                          │
            │                          │
0x0000_0000 └──────────────────────────┘
```

Figure 3-10 Kernel Mode Virtual Address Space

Table 3-8   Kernel Mode Segments

Address Bit Values	Status Register Is One of These Values			Segment Name	Address Range	Segment Size
	UM	EXL	ERL			
A(31) = 0	UM = 0 or EXL = 1 or ERL = 1 and DM = 0			kuseg	0x0000_0000 through 0x7FFF_FFFF	2 Gbytes (2^{31} bytes)
A(31:29) = 100_2				kseg0	0x8000_0000 through 0x9FFF_FFFF	512 Mbytes (2^{29} bytes)
A(31:29) = 101_2				kseg1	0xA000_0000 through 0xBFFF_FFFF	512 Mbytes (2^{29} bytes)
A(31:29) = 110_2				ksseg2	0xC000_0000 through 0xDFFF_FFFF	512 Mbytes (2^{29} bytes)
A(31:29) = 111_2				kseg3	0xE000_0000 through 0xFFFF_FFFF	512 Mbytes (2^{29} bytes)

3.4.2.1 Kernel Mode, User Space (*kuseg*)

In kernel mode, when the most-significant bit of the virtual address (A31) is cleared, the 32-bit *kuseg* virtual address space is selected and covers the full 2^{31} bytes (2 Gbytes) of the current user address space mapped to addresses 0x000_0000 - 0x7FFF_FFFF. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

When *ERL = 1* in the *Status* register, the user address region becomes a 2^{31}-byte unmapped (that is, mapped directly to physical addresses) uncached address space.

3.4.2.2 Kernel Mode, Kernel Space 0 (*kseg0*)

In kernel mode, when the most-significant three bits of the virtual address are 100_2, 32-bit *kseg0* virtual address space is selected; it is the 2^{29}-byte (512-Mbyte) kernel virtual space located at addresses 0x8000_0000 -

0x9FFF_FFFF. References to *kseg0* are not mapped through the TLB; the physical address selected is defined by subtracting 0x8000_0000 from the virtual address. The *K0* field of the *Config* register controls cacheability.

3.4.2.3 Kernel Mode, Kernel Space 1 (*kseg1*)

In kernel mode, when the most-significant three bits of the 32-bit virtual address are 101_2, 32-bit *kseg1* virtual address space is selected; and is the 2^{29}-byte (512-Mbyte) kernel virtual space located at addresses 0xA000_0000 - 0xBFFF_FFFF. References to *kseg1* are not mapped through the TLB; the physical address selected is defined by subtracting 0xA000_0000 from the virtual address. Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly.

3.4.2.4 Kernel Mode, Kernel Space 2 (*kseg2*)

In kernel mode, when *UM = 0, ERL = 1,* or *EXL = 1* in the *Status* register, and *DM = 0* in the Debug register, and the most-significant three bits of the 32-bit virtual address are 110_2, 32-bit *kseg2* virtual address space is selected. In the 4Kp and 4Km processor cores this 2^{29}-byte (512-Mbyte) kernel virtual space is located at addresses 0xC000_0000 - 0xDFFF_FFFF. In the 4Kc processor core this space is mapped through the TLB.

3.4.2.5 Kernel Mode, Kernel Space 3 (*kseg3*)

In kernel mode, when the most-significant three bits of the 32-bit virtual address are 111_2, the *kseg3* virtual address space is selected. In the 4Kp and 4Km processor cores this 2^{29}-byte (512-Mbyte) kernel virtual space is located at addresses 0xE000_0000 - 0xFFFF_FFFF. In the 4Kc processor core this space is mapped through the TLB.

3.4.3 Debug Mode

Debug mode address space is identical to kernel mode address space with respect to unmapped areas. Mapped areas are only accessible if a valid translation is resident in the TLB. In parallel with this, a debug segment *dseg* co-exists in the virtual address range 0xFF20_0000 to 0xFF3F_FFFF. The layout is shown in Figure 3-11.
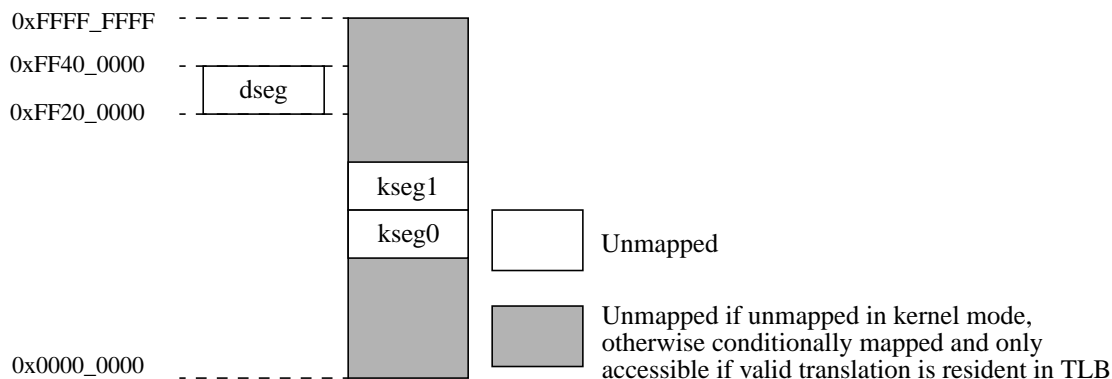
Figure 3-11 Debug Mode Virtual Address Space

Accesses to memory that would normally cause an exception if tried from kernel mode, cause the core to re-enter debug mode via a debug mode exception. This includes accesses usually causing a TLB exception, with the result that such accesses are not handled by the usual memory management routines.

The unmapped kseg0 and kseg1 segments from kernel mode address space are available from debug mode, which allows the debug handler to be executed from uncached and unmapped memory.

The *dseg* is sub-divided into the *dmseg* segment at 0xFF20_0000 to 0xFF2F_FFFF which is used when the probe services the memory segment, and the *drseg* segment at 0xFF30_0000 to 0xFF3F_FFFF which is used when memory mapped debug registers are accessed. The subdivision and attributes for the segments are shown in Table 3-9.

Table 3-9   Physical Address and Cache Attributes for dseg, dmseg, and drseg Address Spaces

Segment Name	Sub-Segment Name	Virtual Address	Generates Physical Address	Cache Attribute
dseg	dmseg	0xFF20_0000 through 0xFF2F_FFFF	dmseg maps to addresses 0x0_0000 - 0xF_FFFF in EJTAG probe memory space.	Uncached
	drseg	0xFF30_0000 through 0xFF3F_FFFF	drseg maps to the breakpoint registers 0x0_0000 - 0xF_FFFF	

3.4.3.1 Conditions and Behavior for Access to drseg, EJTAG registers

The behavior of CPU access to the drseg address range at 0xFF30_0000 to 0xFF3F_FFFF is determined as shown in Table 3-10.

Table 3-10   CPU Access to drseg Address Range

Transaction	LSNM bit in Debug register	Access
Load / Store	1	Kernel mode address space
Fetch	Don't care	drseg, see comments below
Load / Store	0	

Debug software is expected to read the debug control register (DCR) register to determine which other memory mapped registers exist in drseg. The value returned in response to a read of any unimplemented memory mapped register is unpredictable, and writes are ignored to any unimplemented register in the drseg. Refer to Chapter 9 for more information on the DCR.

The allowed access size is limited for the drseg. Only word size transactions are allowed. Operation of the processor is undefined for other transaction sizes.

3.4.3.2 Conditions and Behavior for Access to dmseg, EJTAG memory

The behavior of CPU access to the dmseg address range at 0xFF20_0000 to 0xFF2F_FFFF is determined by the table shown in Table 3-11

Table 3-11   CPU Access to dmseg Address Range

Transaction	ProbEn bit in DCR register	LSNM bit in Debug register	Access
Load / Store	Don't care	1	Kernel mode address space
Fetch	1	Don't care	dmseg
Load / Store	1	0	
Fetch	0	Don't care	See comments below
Load / Store	0	0	

The case with access to the dmseg when the ProbEn bit in the DCR register is 0 is not expected to happen. Debug software is expected to check the state of ProbEn bit in DCR register before attempting to reference dmseg. If such a reference does happen, the reference hangs until it is satisfied by the probe. The probe can not assume that there will never be a reference to dmseg if the ProbEn bit in the DCR register is 0 because there is an inherent race between the debug software sampling the ProbEn bit as 1 and the probe clearing it to 0.

3.5 System Control Coprocessor

The System Control Coprocessor (CP0) is implemented as an integral part of the 4K processor cores and supports memory management, address translation, exception handling, and other privileged operations. Certain CP0 registers are used to support memory management. Refer to Chapter 5 for more information on the CP0 register set.

Exceptions

All MIPS32 4K™ processor cores receive exceptions from a number of sources, including translation lookaside buffer (TLB) misses, arithmetic overflows, I/O interrupts, and system calls. When the CPU detects one of these exceptions, the normal sequence of instruction execution is suspended and the processor enters kernel mode.

In kernel mode the core disables interrupts and forces execution of a software exception processor (called a *handler*) located at a fixed address. The handler saves the context of the processor, including the contents of the program counter, the current operating mode, and the status of the interrupts (enabled or disabled). This context is saved so it can be restored when the exception has been serviced.

When an exception occurs, the core loads the *Exception Program Counter* (*EPC*) register with a location where execution can restart after the exception has been serviced. The restart location in the *EPC* register is the address of the instruction that caused the exception or, if the instruction was executing in a branch delay slot, the address of the branch instruction immediately preceding the delay slot.

This chapter contains the following sections.

- Section 4.1, "Exception Conditions"

- Section 4.2, "Exception Priority"

- Section 4.3, "Exception Vector Locations"

- Section 4.4, "General Exception Processing"

- Section 4.5, "Debug Exception Processing"

- Section 4.6, "Exceptions"

- Section 4.7, "Exception Handling and Servicing Flowcharts"

4.1 Exception Conditions

When an exception condition occurs, the relevant instruction and all those that follow it in the pipeline are cancelled. Accordingly, any stall conditions and any later exception conditions that may have referenced this instruction are inhibited; there is no benefit in servicing stalls for a cancelled instruction.

When an exception condition is detected on an instruction fetch, the core aborts that instruction and all instructions that follow. When this instruction reaches the W stage, the exception flag causes it to write various CP0 registers with the exception state, change the current program counter (PC) to the appropriate exception vector address, and clear the exception bits of earlier pipeline stages.

This implementation allows all preceding instructions to complete execution and prevents all subsequent instructions from completing. Thus, the value in the EPC (ErrorEPC for errors, or DEPC for debug exceptions) is sufficient to restart execution. It also ensures that exceptions are taken in the order of execution; an instruction taking an exception may itself be killed by an instruction further down the pipeline that takes an exception in a later cycle.

4.2 Exception Priority

Table 4-1 lists all possible exceptions, and the relative priority of each, highest to lowest.

Table 4-1   Priority of Exceptions

Exception	Description
Reset	Assertion of SI_ColdReset signal.
Soft Reset	Assertion of SI_Reset signal.
DSS	EJTAG Debug Single Step.
DINT	EJTAG Debug Interrupt. Caused by the assertion of the external EJ_DINT input, or by setting the *EjtagBrk* bit in the ECR register.
NMI	Asserting edge of EB_NMI signal.
Machine Check	TLB write that conflicts with an existing entry (4Kc core).
Interrupt	Assertion of unmasked HW or SW interrupt signal.
Deferred Watch	Deferred Watch (unmasked by K|DM->!(K|DM) transition).
DIB	EJTAG debug hardware instruction break matched.

Table 4-1   Priority of Exceptions

Exception	Description
WATCH	A reference to an address in one of the watch registers (fetch).
AdEL	Fetch address alignment error. Fetch reference to protected address.
TLBL	Fetch TLB miss (4Kc core).
TLBL	Fetch TLB hit to page with V=0 (4Kc core).
IBE	Instruction fetch bus error.
DBp	EJTAG Breakpoint (execution of SDBBP instruction).
Sys	Execution of SYSCALL instruction.
Bp	Execution of BREAK instruction.
CpU	Execution of a coprocessor instruction for a coprocessor that is not enabled.
RI	Execution of a Reserved Instruction.
Ov	Execution of an arithmetic instruction that overflowed.
Tr	Execution of a trap (when trap condition is true).
DDBL / DDBS	EJTAG Data Address Break (address only) or EJTAG Data Value Break on Store (address and value).
WATCH	A reference to an address in one of the watch registers (data).
AdEL	Load address alignment error. Load reference to protected address.
AdES	Store address alignment error. Store to protected address.
TLBL	Load TLB miss (4Kc core).
TLBL	Load TLB hit to page with V=0 (4Kc core).
TLBS	Store TLB miss (4Kc core).
TLBS	Store TLB hit to page with V=0 (4Kc core).
TLB Mod	Store to TLB page with D=0 (4Kc core).

Table 4-1   Priority of Exceptions

Exception	Description
DBE	Load or store bus error.
DDBL	EJTAG data hardware breakpoint matched in load data compare.

4.3 Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location 0xBFC0_0000. Debug exceptions are vectored to location 0xBFC0_0480 or to location 0xFF20_0200 if the ProbTrap bit is 0 or 1, respectively, in the EJTAG Control register (ECR). Addresses for all other exceptions are a combination of a vector offset and a base address. Table 4-2 gives the base address as a function of the exception and whether the BEV bit is set in the *Status* register. Table 4-3 gives the offsets from the base address as a function of the exception. Table 4-4 combines these two tables into one that contains all possible vector addresses as a function of the state that can affect the vector selection.

Table 4-2   Exception Vector Base Addresses

Exception	Status$_{BEV}$	
	0	**1**
Reset, Soft Reset, NMI	0xBFC0_0000	
Debug (with ProbTrap = 0 in the EJTAG Control register)	0xBFC0_0480	
Debug (with ProbTrap = 1 in the EJTAG Control register)	0xFF20_0200 (in dmseg handled by probe, and not system memory)	
Other	0x8000_0000	0xBFC0_0200

Table 4-3   Exception Vector Offsets

Exception	Vector Offset
TLB refill, EXL = 0 (4Kc core)	0x000
Reset, Soft Reset, NMI	0x000 (uses reset base address)
General Exception	0x180
Interrupt, Cause$_{IV}$ = 1	0x200

Table 4-4   Exception Vectors

Exception	BEV	EXL	IV	EJTAG ProbTrap	Vector
Reset, Soft Reset, NMI	x	x	x	x	0xBFC0_0000
Debug	x	x	x	0	0xBFC0_0480
Debug	x	x	x	1	0xFF20_0200 (in dmseg)
TLB Refill (4Kc core)	0	0	x	x	0x8000_0000
TLB Refill (4Kc core)	0	1	x	x	0x8000_0180
TLB Refill (4Kc core)	1	0	x	x	0xBFC0_0200
TLB Refill (4Kc core)	1	1	x	x	0xBFC0_0380
Interrupt	0	0	0	x	0x8000_0180
Interrupt	0	0	1	x	0x8000_0200
Interrupt	1	0	0	x	0xBFC0_0380
Interrupt	1	0	1	x	0xBFC0_0400
All others	0	x	x	x	0x8000_0180
All others	1	x	x	x	0xBFC0_0380
'x' denotes don't care					

4.4 General Exception Processing

With the exception of Reset, Soft Reset, NMI, and Debug exceptions, which have their own special processing as described below, exceptions have the same basic processing flow:

- If the EXL bit in the *Status* register is cleared, the *EPC* register is loaded with the PC at which execution will be restarted and the BD bit is set appropriately in the *Cause* register. If the instruction is not in the delay slot of a branch, the BD bit in *Cause* will be cleared and the value loaded into the *EPC* register is the current PC. If the instruction is in the delay slot of a branch, the BD bit in *Cause* is set and *EPC* is loaded with PC-4. If the EXL bit in the *Status* register is set, the *EPC* register is not loaded and the BD bit is not changed in the *Cause* register.

- The CE and ExcCode fields of the *Cause* registers are loaded with the values appropriate to the exception. The CE field is loaded, but not defined, for any exception type other than a coprocessor unusable exception.

- The EXL bit is set in the *Status* register.

- The processor is started at the exception vector.

The value loaded into EPC represents the restart address for the exception and need not be modified by exception handler software in the normal case. Software need not look at the BD bit in the Cause register unless is wishes to identify the address of the instruction that actually caused the exception.

Note that individual exception types may load additional information into other registers. This is noted in the description of each exception type below.

Operation:

```
if SR_EXL = 0
    if InstructionInBranchDelaySlot then
        EPC <- PC - 4
        Cause_BD <- 1
    else
        EPC <- PC
        Cause_BD <- 0
    endif
    if ExceptionType = TLBRefill then
        vectorOffset <- 0x000
    elseif (ExceptionType = Interrupt) and
        (Cause_IV = 1) then
        vectorOffset <- 0x200
    else
        vectorOffset <- 0x180
    endif
else
```

```
        vectorOffset <- 0x180
endif
Cause_CE <- FaultingCoprocessorNumber
Cause_ExcCode <- ExceptionType
SR_EXL <- 1
if SR_BEV = 1 then
    PC <- 0xBFC0_0200 + vectorOffset
else
    PC <- 0x8000_0000 + vectorOffset
endif
```

4.5 Debug Exception Processing

All debug exceptions have the same basic processing flow:

- The DEPC register is loaded with the program counter (PC) value at which execution will be restarted and the DBD bit is set appropriately in the Debug register. The value loaded into the DEPC register is the current PC if the instruction is not in the delay slot of a branch, or the PC-4 of the branch if the instruction is in the delay slot of a branch.

- The DSS, DBp, DDBL, DDBS, DIB and DINT bits (D* bits at [5:0]) in the Debug register are updated appropriately depending on the debug exception.

- Halt and Doze bits in the Debug register are updated appropriately.

- DM bit in the Debug register is set to 1.

- The processor is started at the debug exception vector.

The value loaded into DEPC represents the restart address for the debug exception and need not be modified by the debug exception handler software in the usual case. Debug software need not look at the DBD bit in the Debug register unless it wishes to identify the address of the instruction that actually caused the debug exception.

A unique debug exception is indicated through the DSS, DBp, DDBL, DDBS, DIB and DINT bits (D* bits at [5:0]) in the Debug register.

No other CP0 registers or fields are changed due to the debug exception, thus no additional state is saved.

Operation:

```
if InstructionInBranchDelaySlot then
    DEPC <- PC-4
    Debug_DBD <- 1
else
```

```
        DEPC <- PC
        Debug_DBD <- 0
    endif
    Debug_D* bits at at [5:0] <- DebugExceptionType
    Debug_Halt <- HaltStatusAtDebugException
    Debug_Doze <- DozeStatusAtDebugException
    Debug_DM <- 1
    if EJTAGControlRegister_ProbTrap = 1 then
        PC <- 0xFF20_0200
    else
        PC <- 0xBFC0_0480
    endif
```

The same debug exception vector location is used for all debug exceptions. The location is determined by the ProbTrap bit in the EJTAG Control register (ECR), as shown in Table 4-5.

Table 4-5   Debug Exception Vector Addresses

ProbTrap bit in ECR Register	Debug Exception Vector Address
0	0xBFC0_0480
1	0xFF20_0200 in dmseg

4.6 Exceptions

The following subsections describe each of the exceptions listed in the same sequence as shown in Table 4-1.

4.6.1 Reset Exception

A reset exception occurs when the SI_ColdReset signal is asserted to the processor. This exception is not maskable. When a Reset exception occurs, the processor performs a full reset initialization, including aborting state machines, establishing critical state, and generally placing the processor in a state in which it can execute instructions from uncached, unmapped address space. On a Reset exception, the state of the processor in not defined, with the following exceptions:

- The *Random* register is initialized to the number of TLB entries - 1.

- The *Wired* register is initialized to zero.

- The *Config* register is initialized with its boot state.

- The RP, BEV, TS, SR, NMI, and ERL fields of the *Status* register are initialized to a specified state.

- The I, R, and W fields of the *WatchLo* register are initialied to 0.

- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC. Note that this value may or may not be predictable.

- PC is loaded with 0xBFC0_0000.

Cause Register ExcCode Value:
    None

Additional State Saved:
    None

Entry Vector Used:
    Reset (0xBFC0_0000)

Operation:

```
Random <- TLBEntries - 1
Wired <- 0
Config <- ConfigurationState
SR_RP <- 0
SR_BEV <- 1
SR_TS <- 0
SR_SR <- 0
SR_NMI <- 0
SR_ERL <- 1
WatchLo_I <- 0
WatchLo_R <- 0
WatchLo_W <- 0
```

```
if InstructionInBranchDelaySlot then
    ErrorEPC <- PC - 4
else
    ErrorEPC <- PC
endif
PC <- 0xBFC0_0000
```

4.6.2 Soft Reset Exception

A soft reset exception occurs when the Reset signal is asserted to the processor. This exception is not maskable. When a soft reset exception occurs, the processor performs a subset of the full reset initialization. Although a soft reset exception does not unnecessarily change the state of the processor, it may be forced to do so in order to place the processor in a state in which it can execute instructions from uncached, unmapped address space. Since bus, cache, or other operations may be interrupted, portions of the cache, memory, or other processor state may be inconsistent. In addition to any hardware initialization required, the following state is established on a soft reset exception:

- The BEV, TS, SR, NMI, and ERL fields of the *Status* register are initialized to a specified state.

- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC. Note that this value may or may not be predictable.

- PC is loaded with 0xBFC0_0000.

Cause **Register ExcCode Value:**
None

Additional State Saved:
None

Entry Vector Used:
Reset (0xBFC0_0000)

Operation:

```
SR_BEV <- 1
SR_TS  <- 0
SR_SR  <- 1
SR_NMI <- 0
SR_ERL <- 1
if InstructionInBranchDelaySlot then
    ErrorEPC <- PC - 4
else
    ErrorEPC <- PC
endif
PC <- 0xBFC0_0000
```

4.6.3 Debug Single Step Exception

A debug single step exception occurs after the CPU has executed one/two instructions in non-debug mode, when returning to non-debug mode after debug mode. One instruction is allowed to execute when returning to a non jump/branch instruction, otherwise two instructions are allowed to execute since the jump/branch and the instruction in the delay slot are executed as one step. Debug single step exceptions are enabled by the SSt bit in the Debug register, and are always disabled for the first one/two instructions after a DERET.

The DEPC register points to the instruction on which the debug single step exception occurred, which is also the next instruction to single step or execute when returning from debug mode. So the DEPC will not point to the instruction which has just been single stepped, but rather the following instruction. The DBD bit in the Debug register is never set for a debug single step exception, since the jump/branch and the instruction in the delay slot is executed in one step.

Exceptions occurring on the instruction(s) executed with debug single step exception enabled are taken even though debug single step was enabled. For a normal exception (other than reset), a debug single step exception is then taken on the first instruction in the normal exception handler. Debug exceptions are unaffected by single step mode, e.g. returning to a SDBBP instruction with debug single step exceptions enabled causes a debug software breakpoint exception, and the DEPC will point to the SDBBP instruction. However, returning to an instruction (not jump/branch) just before the SDBBP instruction, causes a debug single step exception with the DEPC pointing to the SDBBP instruction.

To ensure proper functionality of single step, the debug single step exception has priority over all other exceptions, except reset and soft reset.

Debug Register Debug Status Bit Set
> DSS

Additional State Saved
> None

Entry Vector Used
> Debug exception vector

4.6.4 Debug Interrupt Exception

A debug interrupt exception is either caused by the EjtagBrk bit in the EJTAG Control register (controlled through the TAP), or caused by the debug interrupt request signal to the CPU.

The debug interrupt exception is an asynchronous debug exception which is taken as soon as possible, but with no specific relation to the executed instructions. The DEPC register is set to the instruction where execution should continue after the debug handler is through. The DBD bit is set based on whether the interrupted instruction was executing in the delay slot of a branch.

Debug Register Debug Status Bit Set
    DINT

Additional State Saved
    None

Entry Vector Used
    Debug exception vector

4.6.5 Non Maskable Interrupt (NMI) Exception

A non maskable interrupt exception occurs when the NMI signal is asserted to the processor. NMI is an edge sensitive signal - only one NMI exception will be taken each time NMI is asserted. An NMI exception occurs only at instruction boundaries, so it does not cause any reset or other hardware initialization. The state of the cache, memory, and other processor states are consistent and all registers are preserved, with the following exceptions:

- The BEV, TS, SR, NMI, and ERL fields of the *Status* register are initialized to a specified state.

- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC.

- PC is loaded with 0xBFC0_0000.

Cause **Register ExcCode Value:**
None

Additional State Saved:
None

Entry Vector Used:
Reset (0xBFC0_0000)

Operation:

```
SR_BEV  <- 1
SR_TS   <- 0
SR_SR   <- 0
SR_NMI  <- 1
SR_ERL  <- 1
if InstructionInBranchDelaySlot then
    ErrorEPC <- PC - 4
else
    ErrorEPC <- PC
endif
PC <- 0xBFC0_0000
```

4.6.6 Machine Check Exception (4Kc core)

A machine check exception occurs when the processor detects an internal inconsistency. The following condition causes a machine check exception;

- The detection of multiple matching entries in the TLB in a TLB-based MMU. The core detects this condition on a TLB write and prevents the write from being completed. The TS bit in the *Status* register is set to indicate this condition. This bit is only a status flag and does not affect the operation of the device. Software clears this bit at the appropriate time. This condition is resolved by flushing the conflicting TLB entries. The TLB write can then be completed.

Cause Register ExcCode Value:
MCheck

Additional State Saved:
Depends on the condition that caused the exception.

Entry Vector Used:
General exception vector (offset 0x180)

4.6.7 Interrupt Exception

The interrupt exception occurs when one or more of the eight interrupt requests is enabled by the *Status* register and the interrupt input is asserted.

Register ExcCode Value:
Int

Additional State Saved:

Table 4-6   Register States an Interrupt Exception

Register State	Value
$Cause_{IP}$	indicates the interrupts that are pending.

Entry Vector Used:
General exception vector (offset 0x180) if the IV bit in the *Cause* register is 0;
interrupt vector (offset 0x200) if the IV bit in the *Cause* register is 1.

4.6.8 Debug Instruction Break Exception

A debug instruction break exception occurs when an instruction hardware breakpoint matches an executed instruction. The DEPC register and DBD bit in the Debug register indicates the instruction that caused the instruction hardware breakpoint to match. This exception can only occur if instruction hardware breakpoints are implemented.

Debug Register Debug Status Bit Set:
   DIB

Additional State Saved:
   None

Entry Vector Used:
   Debug exception vector

4.6.9 Watch Exception — Instruction Fetch or Data Access

The Watch facility provides a software debugging vehicle by initiating a watch exception when an instruction or data reference matches the address information stored in the *WatchHi* and *WatchLo* registers. A Watch exception is taken immediately if the EXL and ERL bits of the *Status* register are both zero. If either bit is a one at the time that a watch exception would normally be taken, the WP bit in the *Cause* register is set, and the exception is deferred until both the EXL and ERL bits in the Status register are zero. Software may use the WP bit in the *Cause* register to determine if the EPC register points at the instruction that caused the watch exception, or if the exception actually occurred while in kernel mode.

The Watch exception can occur on either an instruction fetch or a data access. Watch exceptions that occur on an instruction fetch have a higher priority than watch exceptions that occur on a data access.

Register ExcCode Value:
   WATCH

Additional State Saved:

Table 4-7   Register States on a Watch Exception

Register State	Value
Cause$_{WP}$	Indicates that the watch exception was deferred until after both Status$_{EXL}$ and Status$_{ERL}$ were zero. This bit directly causes a watch exception, so software must clear this bit as part of the exception handler to prevent a watch exception loop at the end of the current handler execution.

Entry Vector Used:
   General exception vector (offset 0x180)

4.6.10 Address Error Exception — Instruction Fetch/Data Access

An address error exception occurs on an instruction or data access when an attempt is made to execute one of the following:

- Fetch an instruction, load a word, 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 the kernel address space from user mode

Note that in the case of an instruction fetch that is not aligned on a word boundary, PC is updated before the condition is detected. Therefore, both EPC and BadVAddr point to the unaligned instruction address. In the case of a data access the exception is taken if either an unaligned address or an address that was inaccessible in the current processor mode was referenced by a load or store instruction.

Cause Register ExcCode Value:
ADEL: Reference was a load or an instruction fetch
ADES: Reference was a store

Additional State Saved:

Table 4-8   CP0 Register States on an Address Exception Error

Register State	Value
BadVAddr	failing address
Context$_{VPN2}$	UNPREDICTABLE
EntryHi$_{VPN2}$	UNPREDICTABLE
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:
General exception vector (offset 0x180)

4.6.11 TLB Refill Exception — Instruction Fetch or Data Access (4Kc core)

During an instruction fetch or data access, a TLB refill exception occurs when no TLB entry in a TLB-based MMU matches a reference to a mapped address space and the EXL bit is 0 in the *Status* register. Note that this is distinct from the case in which an entry matches but has the valid bit off. In that case, a TLB Invalid exception occurs.

Cause **Register ExcCode Value:**
TLBL: Reference was a load or an instruction fetch
TLBS: Reference was a store

Additional State Saved:

Table 4-9   CP0 Register States on a TLB Refill Exception

Register State	Value
BadVAddr	failing address
Context	The BadVPN2 fields contains $VA_{31:13}$ of the failing address
EntryHi	The VPN2 field contains $VA_{31:13}$ of the failing address; the ASID field contains the ASID of the reference that missed
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:
TLB refill vector (offset 0x000) if $Status_{EXL} = 0$ at the time of exception;
general exception vector (offset 0x180) if $Status_{EXL} = 1$ at the time of exception

4.6.12 TLB Invalid Exception — Instruction Fetch or Data Access (4Kc core)

During an instruction fetch or data access, a TLB invalid exception occurs in one of the following cases:

- No TLB entry in a TLB-based MMU matches a reference to a mapped address space; and the EXL bit is 1 in the *Status* register.

- A TLB entry in a TLB-based MMU matches a reference to a mapped address space, but the matched entry has the valid bit off.

- The virtual address is greater than or equal to the bounds address in a BAT-based MMU.

Cause Register ExcCode Value:
TLBL: Reference was a load or an instruction fetch
TLBS: Reference was a store

Additional State Saved:

Table 4-10   CP0 Register States on a TLB Invalid Exception

Register State	Value
BadVAddr	failing address
Context	The BadVPN2 field contains $VA_{31:13}$ of the failing address
EntryHi	The VPN2 field contains $VA_{31:13}$ of the failing address; the ASID field contains the ASID of the reference that missed
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:
General exception vector (offset 0x180)

4.6.13 Bus Error Exception — Instruction Fetch or Data Access

A bus error exception occurs when an instruction or data access makes a bus request (due to a cache miss or an uncacheable reference) and that request terminates in an error. The bus error exception can occur on either an instruction fetch or a data access. Bus error exceptions that occur on an instruction fetch have a higher priority than bus error exceptions that occur on a data access.

Bus errors taken on the requested (critical) word of an instruction fetch or data load are precise. Other bus errors, such as stores or non-critical words of a burst read, can be imprecise. These errors are taken when the EB_RBErr or EB_WBErr signals are asserted and may occur on an instruction that was not the source of the offending bus cycle.

***Cause* Register ExcCode Value:**
> IBE: Error on an instruction reference
> DBE: Error on a data reference

Additional State Saved:
> None

Entry Vector Used:
> General exception vector (offset 0x180)

4.6.14 Debug Software Breakpoint Exception

A debug software breakpoint exception occurs when a SDBBP instruction is executed. The DEPC register and DBD bit in the Debug register will indicate the SDBBP instruction that caused the debug exception.

Debug Register Debug Status Bit Set:
DBp

Additional State Saved:
None

Entry Vector Used:
Debug exception vector

4.6.15 Execution Exception — System Call

The system call exception is one of the six execution exceptions. All of these exceptions have the same priority. A system call exception occurs when a SYSCALL instruction is executed.

Cause **Register ExcCode Value:**

Sys

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.6.16 Execution Exception — Breakpoint

The breakpoint exception is one of the six execution exceptions. All of these exceptions have the same priority. A breakpoint exception occurs when a BREAK instruction is executed.

Cause Register ExcCode Value:
Bp

Additional State Saved:
None

Entry Vector Used:
General exception vector (offset 0x180)

4.6.17 Execution Exception — Reserved Instruction

The reserved instruction exception is one of the six execution exceptions. All of these exceptions have the same priority. A reserved instruction exception occurs when a reserved or undefined major opcode or function field is executed.

Cause **Register ExcCode Value:**
RI

Additional State Saved:
None

Entry Vector Used:
General exception vector (offset 0x180)

4.6.18 Execution Exception — Coprocessor Unusable

The coprocessor unusable exception is one of the six execution exceptions. All of these exceptions have the same priority. A coprocessor unusable exception occurs when an attempt is made to execute a coprocessor instruction for one of the following:

- a corresponding coprocessor unit that has not been marked usable by setting its CU bit in the *Status* register

- CP0 instructions, when the unit has not been marked usable, and the processor is executing in user mode

Cause **Register ExcCode Value:**
    CpU

Additional State Saved:

Table 4-11   Register States on a Coprocessor Unusable Exception

Register State	Value
Cause$_{CE}$	unit number of the coprocessor being referenced

Entry Vector Used:
    General exception vector (offset 0x180)

4.6.19 Execution Exception — Integer Overflow

The integer overflow exception is one of the six execution exceptions. All of these exceptions have the same priority. An integer overflow exception occurs when selected integer instructions result in a 2's complement overflow.

Cause **Register ExcCode Value:**
    Ov

Additional State Saved:
    None

Entry Vector Used:
    General exception vector (offset 0x180)

4.6.20 Execution Exception — Trap

The trap exception is one of the six execution exceptions. All of these exceptions have the same priority. A trap exception occurs when a trap instruction results in a TRUE value.

Cause **Register ExcCode Value:**
Tr

Additional State Saved:
None

Entry Vector Used:
General exception vector (offset 0x180)

4.6.21 Debug Data Break Exception

A debug data break exception occurs when a data hardware breakpoint matches the load/store transaction of an executed load/store instruction. The DEPC register and DBD bit in the Debug register will indicate the load/store instruction that caused the data hardware breakpoint to match. The load/store instruction that caused the debug exception has not completed e.g. not updated the register file, and the instruction can be re-executed after returning from the debug handler.

Debug Register Debug Status Bit Set:
DDBL for a load instruction or DDBS for a store instruction

Additional State Saved:
None

Entry Vector Used:
Debug exception vector

4.6.22 TLB Modified Exception — Data Access (4Kc core)

During a data access, a TLB modified exception occurs on a *store* reference to a mapped address if the following condition is true:

- The matching TLB entry in a TLB-based MMU is valid, but not dirty.

Cause Register ExcCode Value:
Mod

Additional State Saved:

Table 4-12   Register States on a TLB Modified Exception

Register State	Value
BadVAddr	failing address
Context	The BadVPN2 field contains $VA_{31:13}$ of the failing address.
EntryHi	The VPN2 field contains $VA_{31:13}$ of the failing address; the ASID field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:
General exception vector (offset 0x180)

4.7 Exception Handling and Servicing Flowcharts

The remainder of this chapter contains flowcharts for the following exceptions and guidelines for their handlers:

- General exceptions and their exception handler

- TLB miss exception and their exception handler

- Reset, soft reset and NMI exceptions, and a guideline to their handler.

- Debug exceptions

Generally speaking, the exceptions are handled by hardware (HW); the exceptions are then serviced by software (SW). Note that unexpected debug exceptions to the debug exception vector at 0xBFC0_0200 may be viewed as a reserved instruction since uncontrolled execution of a SDBBP instruction caused the exception. The DERET instruction must be used at return from the debug exception handler, in order to leave debug mode and return to non-debug mode. The DERET instruction returns to the address in the DEPC register.

Exceptions other than Reset, Soft Reset, NMI, or first-level miss
Note: Interrupts can be masked by IE or IMs
and Watch is masked if EXL = 1

Comments

EnHi <- VPN2, ASID
Context <- VPN2
Set Cause Register
 EXCCode, CE
 Set BadVA

EnHi and Context are set only for
*TLB- Invalid, Modified,
& Refill exceptions.
BadVA is set only for
TLB- Invalid, Modified,
Refill- and VCED/I exceptions
 Note: not set if it is a Bus Error

Check if exception within
another exception

EXL
(SR1)

=1

=0

Instr. in
Br.Dly. Slot?

Yes

No

Cause 31 (BD) <- 1
EPC <- (PC - 4)

Cause 31 (BD) <- 0
EPC <- PC

EXL <- 1

Processor forced to Kernel Mode
& interrupt disabled

=0 (normal)

BEV

=1 (bootstrap)

PC <- 0x8000_0000 + 180
(unmapped, cached)

PC <- 0xBFC0_0200 + 180
(unmapped, uncached)

To General Exception Servicing Guidelines

Figure 4-1   General Exception Handler (HW)

Comments

MFC0 -
Context
EPC
Status
Cause

* Unmapped vector so TLBMod, TLBInv,
  TLB Refill exceptions not possible

* EXL=1 so Watch, Interrupt exceptions disabled

* OS/System to avoid all other exceptions

*Only CacheError, Reset, Soft Reset, NMI
  exceptions possible.

MTC0 -
(Set Status Bits:)
UM <- 0
EXL <- 0
& IE=1

(Optional - only to enable Interrupts while keeping Kernel Mode)

Check Cause value & Jump to
appropriate Service Code

* After EXL=0, all exceptions allowed.
  (except interrupt if masked by IE or IM
  and CacheError if masked by DE)

Status
bit 21(TS)

=1

=0

Reset the processor

Service Code

EXL = 1

MTC0 -
EPC
STATUS

ERET

* ERET is not allowed in the branch delay slot of
  another Jump Instruction

* Processor does not execute the instruction which is
  in the ERET's branch delay slot

* PC <- EPC; EXL <- 0

* LLbit <- 0

Figure 4-2   General Exception Servicing Guidelines (SW)
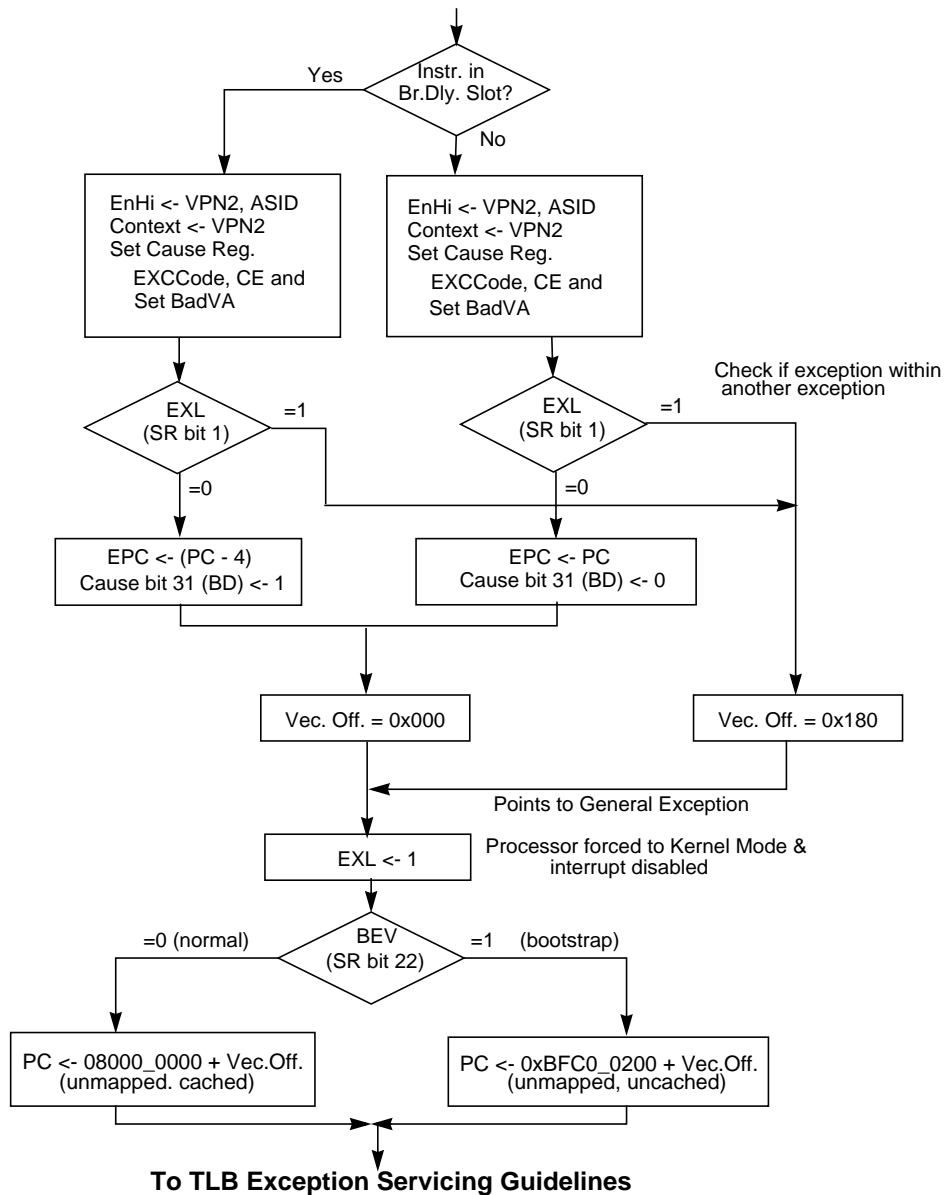
Figure 4-3   TLB Miss Exception Handler (HW) — 4Kc Core

Comments

MFC0 -

CONTEXT

* Unmapped vector so TLBMod, TLBInv,
TLB Refill or VCEP exceptions
not possible

* EXL=1 so Watch, Interrupt exceptions disabled

* OS/System to avoid all other exceptions

*Only Reset, Soft Reset, NMI
 exceptions possible.

Service Code

* Load the mapping of the virtual address in Context Reg.
Move it to ENLO and Write into the TLB

* There could be a TLB miss again during the mapping
of the data or instruction address. The processor will
jump to the general exception vector since the EXL is 1.
(Option to complete the first level refill in the general
exception handler or ERET to the original instruction
and take the exception again)

ERET

* ERET is not allowed in the branch delay slot of
another Jump Instruction

* Processor does not execute the instruction which is
in the ERET's branch delay slot

* PC <- EPC; EXL <- 0

* LLbit <- 0

Figure 4-4   TLB Exception Servicing Guidelines (SW) — 4Kc and 4Km Cores

Soft Reset or NMI Exception

Status:
 BEV <- 1
 TS <- 0
 SR<- 1
 ERL <- 1

Reset Exception

Random <- TLBENTRIES - 1

Wired <- 0

Config <- Update(31:6)|| Undef(5:0)

Status:
 RP <- 0
 BEV <- 1
 TS <- 0
 SR<- 0
 ERL <- 1

WatchLo:
 I,R,W <- 0

ErrorEPC <- PC

PC <- 0xBFC0_0000

Reset, Soft Reset & NMI Exception Handling (HW)

Reset, Soft Reset & NMI Servicing Guidelines (SW)

Status bit 19 (NMI)

= 1

= 0

Status bit 20 (SR)

=0

= 1

NMI Service Code

ERET

(Optional)

Soft Reset Service Code

Reset Service Code

Figure 4-5   Reset, Soft Reset and NMI Exception Handling and Servicing Guidelines

CP0 Registers

The System Control Coprocessor (CP0) provides the register interface to the MIPS32 4K™ processor cores and supports memory management, address translation, exception handling, and other privileged operations. Each CP0 register has a unique number that identifies it; this number is referred to as the *register number.* For instance, the *PageMask* register is register number 5. For more information on the EJTAG registers, refer to Chapter 9. This chapter contains the following sections:

- Section 5.1, "CP0 Register Summary"

- Section 5.2, "CP0 Registers"

5.1 CP0 Register Summary

Table 5-1 lists the CP0 registers in numerical order. The individual registers are described throughout this chapter.

Table 5-1   CP0 Registers

Register Number	Register Name	Function
0	Index[1]	Index into the TLB array (4Kc core). This register is reserved in the 4Kp and 4Km cores.
1	Random[1]	Randomly generated index into the TLB array (4Kc core). This register is reserved in the 4Kp and 4Km cores.
2	EntryLo0[1]	Low-order portion of the TLB entry for even-numbered virtual pages (4Kc core). This register is reserved in the 4Kp and 4Km cores.
3	EntryLo1[1]	Low-order portion of the TLB entry for odd-numbered virtual pages (4Kc core). This register is reserved in the 4Kp and 4Km cores.
4	Context[2]	Pointer to page table entry in memory (4Kc core). This register is reserved in the 4Kp and 4Km cores.
5	PageMask[1]	Controls the variable page sizes in TLB entries (4Kc core). This register is reserved in the 4Kp and 4Km cores.
6	Wired[1]	Controls the number of fixed ("wired") TLB entries (4Kc core). This register is reserved in the 4Kp and 4Km cores.
7	Reserved	Reserved
8	BadVAddr[2]	Reports the address for the most recent address-related exception
9	Count[2]	Processor cycle count
10	EntryHi[1]	High-order portion of the TLB entry (4Kc core). This register is reserved in the 4Kp and 4Km cores.
11	Compare[2]	Timer interrupt control
12	Status[2]	Processor status and control
13	Cause[2]	Cause of last exception

Table 5-1   CP0 Registers (continued)

Register Number	Register Name	Function
14	EPC[2]	Program counter at last exception
15	PRId	Processor identification and revision
16	Config/Config1	Configuration register
17	LLAddr	Load linked address
18	WatchLo[2]	Low-order watchpoint address
19	WatchHi[2]	High-order watchpoint address
20 - 22	Reserved	Reserved
23	Debug[3]	Debug control and exception status
24	DEPC[3]	Program counter at last debug exception
25 - 27	Reserved	Reserved
28	TagLo/DataLo	Low-order portion of cache tag interface
29	Reserved	Reserved
30	ErrorEPC[2]	Program counter at last error
31	DESAVE[3]	Debug handler scratch pad register
1. Registers used in memory management. 2. Registers used in exception processing. 3. Registers used in debug.		

5.2 CP0 Registers

The CP0 registers provide the interface between the ISA and the architecture. Each register is discussed below, with the registers presented in numerical order, first by register number, then by select field number.

For each register described below, field descriptions include the read/write properties of the field, and the reset state of the field. For the read/write properties of the field, the following notation is used:

Table 5-2   CP0 Register Field Types

Read/Write Notation	Hardware Interpretation	Software Interpretation
R/W	A field in which all bits are readable and writable by software and, potentially, by hardware. Hardware updates of this field are visible by software read. Software updates of this field are visible by hardware read. If the reset state of this field is "Undefined," either software or hardware must initialize the value before the first read will return a predictable value. This should not be confused with the formal definition of UNDEFINED behavior.	
R	A field that is either static or is updated only by hardware. If the Reset State of this field is either "0" or "Preset", hardware initializes this field to zero or to the appropriate state, respectively, on powerup. If the Reset State of this field is "Undefined", hardware updates this field only under those conditions specified in the description of the field.	A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware. If the Reset State of this field is "Undefined," software reads of this field result in an UNPREDICTABLE value except after a hardware update done under the conditions specified in the description of the field.
0	A field that hardware does not update, and for which hardware can assume a zero value.	A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in UNDEFINED behavior of the hardware. Software reads of this field return zero as long as all previous software writes are zero. If the Reset State of this field is "Undefined," software must write this field with zero before it is guaranteed to read as zero.

5.2.1 Index Register (CP0 Register 0, Select 0)

The *Index* register is a 32-bit read/write register that contains the index used to access the TLB for TLBP, TLBR, and TLBWI instructions. The width of the index field is implementation-dependent as a function of the number of TLB entries that are implemented. The minimum value for TLB-based MMUs is *Ceiling(Log2(TLBEntries))*.

The operation of the processor is UNDEFINED if a value greater than or equal to the number of TLB entries is written to the *Index* register.

This register is only valid with the TLB (4Kc core). It is reserved if the BAT is implemented (4Kp and 4Km).

Index Register Format

31	30		4	3	0
P		0		Index	

Table 5-3   Index Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
P	31	Probe Failure. Set to 1 when the previous TLBProbe (TLBP) instruction failed to find a match in the TLB.	R	Undefined
0	30:4	Must be written as zero; returns zero on read.	0	0
Index	3:0	Index to the TLB entry affected by the TLBRead and TLBWrite instructions.	R/W	Undefined

5.2.2 Random Register (CP0 Register 1, Select 0)

The *Random* register is a read-only register whose value is used to index the TLB during a TLBWR instruction. The width of the Random field is calculated in the same manner as that described for the *Index* register above.

The value of the register varies between an upper and lower bound as follow:

- A lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the *Wired* register). The entry indexed by the *Wired* register is the first entry available to be written by a TLB Write Random operation.

- An upper bound is set by the total number of TLB entries minus 1.

The Random register is decremented by one every clock until the value in the *Wired* register is reached. To enhance the level of randomness and reduce the possibility of a live lock condition, an LFSR register is used that prevents the decrement pseudo-randomly.

The processor initializes the *Random* register to the upper bound on a Reset exception and when the *Wired* register is written.

This register is only valid with the TLB (4Kc core). It is reserved if the BAT is implemented (4Kp and 4Km).

Random Register Format

31	4	3	0
0		Random	

Table 5-4   Random Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
0	31:4	Must be written as zero; returns zero on read.	0	0
Random	3:0	TLB Random Index	R	TLB Entries - 1

5.2.3 EntryLo0, EntryLo1 (CP0 Registers 2 and 3, Select 0)

The pair of EntryLo registers act as the interface between the TLB or BAT and the TLBR, TLBWI, and TLBWR instructions. For a TLB-based MMU, EntryLo0 holds the entries for even pages and EntryLo1 holds the entries for odd pages. For a BAT-based MMU, only EntryLo0 is used to hold the base information for the BAT entry.

The contents of the EntryLo0 and EntryLo1 registers are undefined after an address error, TLB invalid, TLB modified, or TLB refill exceptions.

These registers are only valid with the TLB (4Kc core). They are reserved if the BAT is implemented (4Kp and 4Km).

EntryLo0, EntryLo1 Register Format

31 30 29	26 25	6 5	3 2 1 0
R	0	PFN	C	D	V	G

Table 5-5   EntryLo0, EntryLo1 Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
R	31:30	Reserved. Should be ignored on writes; returns zero on read.	R	0
0	29:26	These 4 bits are normally part of the PFN. However, since the core supports only 32-bits of physical address, the PFN is only 20-bits wide. Therefore, bits 29:26 of this register must be written with zeros.	R/W	0
PFN	25:6	Page Frame Number. Corresponds to bits 31:12 of the physical address.	R/W	Undefined
C	5:3	Coherency attribute of the page. See Table 5-6.	R/W	Undefined
D	2	"Dirty" or write-enable bit, indicating that the page has been written, and/or is writable. If this bit is a one, stores to the page are permitted. If this bit is a zero, stores to the page cause a TLB Modified exception.	R/W	Undefined

Table 5-5   EntryLo0, EntryLo1 Register Field Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
V	1	Valid bit, indicating that the TLB entry, and thus the virtual page mapping are valid. If this bit is a one, accesses to the page are permitted. If this bit is a zero, accesses to the page cause a TLB Invalid exception.	R/W	Undefined
G	0	Global bit. On a TLB write, the logical AND of the G bits in both the EntryLo0 and EntryLo1 registers become the G bit in the TLB entry. If the TLB entry G bit is a one, ASID comparisons are ignored during TLB matches. On a read from a TLB entry, the G bits of both EntryLo0 and EntryLo1 reflect the state of the TLB G bit.	R/W	Undefined

Table 5-6 lists the encoding of the C field of the *EntryLo0* and *EntryLo1* registers and the K0 field of the *Config* register.

Table 5-6   Cache Coherency Attributes

C(5:3) Value	Cache Coherency Attribute
0, 1, 3*, 4, 5, 6	Cacheable, noncoherent, write through, no write allocate
2*, 7	Uncached

* These two values are required by the MIPS32 architecture. All other values are not used. For example, values 0, 1, 4, 5 and 6 are not used and are mapped to 3. The value 7 is not used and is mapped to 2.

Note that these values do have meaning in other MIPS Technologies processor implementations. Refer to the MIPS32 specification for more information.

5.2.4 Context Register (CP0 Register 4, Select 0)

The *Context* register is a read/write register containing a pointer to an entry in the page table entry (PTE) array. This array is an operating system data structure that stores virtual-to-physical translations. During a TLB miss, the operating system loads the TLB with the missing translation from the PTE array. The *Context* register duplicates some of the information provided in the *BadVAddr* register but is organized in such a way that the operating system can directly reference an 8-byte page table entry (PTE) in memory.

A TLB exception (TLB Refill, TLB Invalid, or TLB Modified) causes bits $VA_{31:13}$ of the virtual address to be written into the *BadVPN2* field of the *Context* register. The *PTEBase* field is written and used by the operating system.

The BadVPN2 field of the *Context* register is not defined after an address error exception.

Context Register Format

31 23	22 4	3 2 1 0
PTEBase	BadVPN2	0

Table 5-7   Context Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
PTEBase	31:23	This field is for use by the operating system and is normally written with a value that allows the operating system to use the *Context* Register as a pointer into the current PTE array in memory.	R/W	Undefined
BadVPN2	22:4	This field is written by hardware on a TLB miss. It contains bits $VA_{31:13}$ of the virtual address that missed.	R	Undefined
0	3:0	Must be written as zero; returns zero on read.	0	0

5.2.5 PageMask Register (CP0 Register 5, Select 0)

The *PageMask* register is a read/write register used for reading from and writing to the TLB. It holds a comparison mask that sets the variable page size for each TLB entry as shown in Table 5-9. Behavior is **UNDEFINED** if a value other than those listed is used.

This register is only valid with the TLB (4Kc core). It is reserved if the BAT is implemented (4Kp and 4Km).

PageMask Register Format

31	25	24		13	12		0
	0		Mask			0	

Table 5-8   PageMask Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Mask	24:13	The Mask field is a bit mask in which a "1" indicates that the corresponding bit of the virtual address should not participate in the TLB match.	R/W	Undefined
0	31:25, 12:0	Must be written as zero; returns zero on read.	0	0

Table 5-9   Values for the Mask Field of the PageMask Register

Page Size	Bit											
	24	23	22	21	20	19	18	17	16	15	14	13
4 KBytes	0	0	0	0	0	0	0	0	0	0	0	0
16 KBytes	0	0	0	0	0	0	0	0	0	0	1	1
64 KBytes	0	0	0	0	0	0	0	0	1	1	1	1
256 KBytes	0	0	0	0	0	0	1	1	1	1	1	1
1 MByte	0	0	0	0	1	1	1	1	1	1	1	1
4 MByte	0	0	1	1	1	1	1	1	1	1	1	1
16 Mbyte	1	1	1	1	1	1	1	1	1	1	1	1

5.2.6 Wired Register (CP0 Register 6, Select 0)

The *Wired* register is a read/write register that specifies the boundary between the wired and random entries in the TLB as shown in Figure 5-1. The width of the Wired field is calculated in the same manner as that described for the *Index* register above. Wired entries are fixed, non-replaceable entries that are not overwritten by a TLBWR instruction. Wired entries can be overwritten by a TLBWI instruction.

The *Wired* register is set to zero by a Reset exception. Writing the *Wired* register causes the *Random* register to reset to its upper bound.

The operation of the processor is undefined if a value greater than or equal to the number of TLB entries is written to the *Wired* register.

This register is only valid with a TLB (4Kc core). It is reserved if the BAT is implemented (4Kp and 4Km cores).
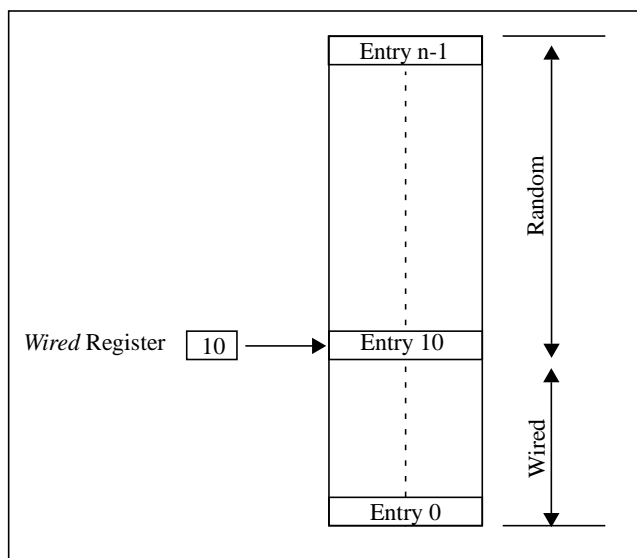
Figure 5-1   Wired and Random Entries in the TLB

Wired Register Format

31	4	3	0
0		Wired	

Table 5-10   Wired Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
0	31:4	Must be written as zero; returns zero on read.	0	0
Wired	3:0	TLB wired boundary.	R/W	0

5.2.7 BadVAddr Register (CP0 Register 8, Select 0)

The *BadVAddr* register is a read-only register that captures the most recent virtual address that caused one of the following exceptions:

- Address error (AdEL or AdES)

- TLB Refill (4Kc core)

- TLB Invalid (4Kc core)

- TLB Modified (4Kc core)

The *BadVAddr* register does not capture address information for cache or bus errors, since neither is an addressing error.

BadVAddr Register Format

31 0
BadVAddr

Table 5-11 BadVAddr Register Field Description

Fields		Description	Read/ Write	Reset State
Name	Bits			
BadVAddr	31:0	Bad virtual address	R	Undefined

5.2.8 Count Register (CP0 Register 9, Select 0)

The Count register acts as a timer, incrementing at a constant rate, whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. The counter increments every other clock.

The Count register can be written for functional or diagnostic purposes, including at reset or to synchronize processors.

The Count register continues incrementing while the processor is in debug mode.

Count Register Format

31	0
Count	

Table 5-12   Count Register Field Description

Fields		Description	Read/ Write	Reset State
Name	Bits			
Count	31:0	Interval counter.	R/W	Undefined

5.2.9 EntryHi Register (CP0 Register 10, Select 0)

The *EntryHi* register contains the virtual address match information used for TLB read, write, and access operations.

A TLB exception (TLB Refill, TLB Invalid, or TLB Modified) causes bits $VA_{31:13}$ of the virtual address to be written into the VPN2 field of the *EntryHi* register. The ASID field is written by software with the current address space identifier value and is used during the TLB comparison process to determine TLB match. The ASID field is not implemented in a BAT-based MMU.

The VPN2 field of the *EntryHi* register is not defined after an address error exception.

This register is only valid with the TLB (4Kc core). It is reserved if the BAT is implemented (4Kp and 4Km cores).

EntryHi Register Format

31	13 12	8 7	0
VPN2	0	ASID	

Table 5-13   EntryHi Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
VPN2	31:13	$VA_{31:13}$ of the virtual address (virtual page number / 2). This field is written by hardware on a TLB exception or on a TLB read, and is written by software before a TLB write.	R/W	Undefined
0	12:8	Must be written as zero; returns zero on read.	0	0
ASID	7:0	Address space identifier. This field is written by hardware on a TLB read and by software to establish the current ASID value for TLB write and against which TLB references match each entry's TLB ASID field. For a BAT-based MMU, this field must be written as zero and returns zero on read.	R/W	Undefined

5.2.10 Compare Register (CP0 Register 11, Select 0)

The *Compare* register acts in conjunction with the *Count* register to implement a timer and timer interrupt function. The timer interrupt is an output of the cores. The *Compare* register maintains a stable value and does not change on its own.

When the value of the *Count* register equals the value of the *Compare* register, the SI_TimerInt pin is asserted. This pin will remain asserted until the *Compare* register is written. The SI_TimerInt pin can be fed back into the core on one of the interrupt pins to generate an interrupt. Traditionally, this has been done by multiplexing it with hardware interrupt 5 to set interrupt bit IP(7) in the *Cause* register.

For diagnostic purposes, the *Compare* register is a read/write register. In normal use, however, the *Compare* register is write-only. Writing a value to the *Compare* register, as a side effect, clears the timer interrupt.

Compare Register Format

31 0
Compare

Table 5-14   Compare Register Field Description

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Compare	31:0	Interval count compare value	R/W	Undefined

5.2.11 Status Register (CP0 Register 12, Select 0)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. Fields of this register combine to create operating modes for the processor, as follows:

Interrupt Enable: Interrupts are enabled when all of the following conditions are true:

- IE = 1
- EXL = 0
- ERL = 0
- DM = 0

If these conditions are met, the settings of the IM and IE bits enable the interrupt.

Operating Modes: If the DM bit in the Debug register is 1, the processor is in debug mode. Otherwise the processor is in either kernel or user mode. The following CPU Status register bit settings determine user or kernel mode.

- User mode: UM = 1, EXL = 0, and ERL = 0
- Kernel mode: UM = 0, or EXL = 1, or ERL = 1

Coprocessor Accessibility: The Status register CU bits control coprocessor accessibility. If any coprocessor is unusable, an instruction that accesses it generates an exception.

Coprocessor 0 is always enabled in kernel mode, regardless of the setting of the CU0 bit.

Status Register Format

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	8	7	5	4	3	2	1	0
CU3-CU0		RP	R	RE		0	BEV	TS	SR	NMI	0		0	IM7-IM0		R		UM	R	ERL	EXL	IE

Table 5-15   Status Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
CU3-CU0	31:28	Controls access to coprocessors 3, 2, 1, and 0, respectively: 0: access not allowed 1: access allowed Coprocessor 0 is always usable when the processor is running in kernel mode, independent of the state of the CU0 bit. The core does not support coprocessors 1-3, but CU3:1 can still be set. However, processor behavior is unpredictable if a coprocessor instruction to coprocessors 1-3 is attempted with the corresponding CU3:1 bit set.	R/W	Undefined
RP	27	Enables reduced power mode. The state of the RP bit is available on the bus interface as the SI_RP signal.	R/W	0 for Cold Reset only.
R	26	This bit must be ignored on write and read as zero.	R	0
RE	25	Used to enable reverse-endian memory references while the processor is running in user mode: 0: User mode uses configured endianness 1: User mode uses reversed endianness Kernel or debug mode references are not affected by the state of this bit.	R/W	Undefined
0	24:23	This bit must be written as zero; returns zero on read.	0	Undefined
BEV	22	Controls the location of exception vectors: 0: Normal 1: Bootstrap	R/W	1

Table 5-15   Status Register Field Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
TS	21	TLB shutdown. This bit is set if a TLBWI or TLBWR instruction is issued that would cause a TLB shutdown condition if allowed to complete. This bit is only used in the 4Kc processor and is reserved in the 4Kp and 4Km processors. Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.	R/W	0

SR	20	Indicates that the entry through the reset exception vector was due to a Soft Reset:
		
		  0: Not Soft Reset (NMI or hard reset)
		  1: Soft Reset
		
		Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.
			R/W	1 for Soft Reset; 0 otherwise

NMI	19	Indicates that the entry through the reset exception vector was due to an NMI.
		
		  0: Not NMI (soft or hard reset)
		  1: NMI
		
		Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.
			R/W	1 for NMI; 0 otherwise

0	18	Must be written as zero; returns zero on read.	0	Undefined
R	17:16	Reserved. Must be ignored on write and read as zero.		Undefined

IM[7:0]	15:8	Interrupt Mask: Controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled and the corresponding bits are set in both the Interrupt Mask field of the Status register and the Interrupt Pending field of the Cause register and the IE bit is set in the Status register.
		
		  0: Interrupt request disabled
		  1: Interrupt request enabled
			R/W	Undefined

R	7:5	Reserved. Must be ignored on write and read as zero.	R	0

Table 5-15   Status Register Field Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			

UM	4	Indicates that the processor is operating in user mode:
		
		  0: processor is operating in kernel mode
		  1: processor is operating in user mode
		
		Note that the processor can also be in kernel mode if EXR or ERL are set. This condition does not affect the state of the UM bit.
			R/W	Undefined

R	3	Reserved. Must be ignored on write and read as zero.	R	0

ERL	2	Error Level. Set by the processor when a Reset, Soft Reset, or NMI exceptions is taken.
		
		  0: normal level
		  1: error level
		
		When ERL is set:
		
		The processor is running in kernel mode.
		
		Interrupts are disabled.
		
		The ERET instruction uses the return address held in ErrorEPC instead of EPC.
		
		kuseg is treated as an unmapped and uncached region. This allows main memory to be accessed in the presence of cache errors.Behavior is **UNDEFINED** if ERL is set while executing code in useg/kuseg.
			R/W	1

EXL	1	Exception Level. Set by the processor when any exception other than Reset, Soft Reset, or NMI exceptions is taken.
		
		  0: normal level
		  1: exception level
		
		When EXL is set:
		
		The processor is running in kernel mode.
		
		Interrupts are disabled.
		
		In the 4Kc core, TLB refill exceptions use the general exception vector instead of the TLB refill vector.
		
		EPC is not updated if another exception is taken.
			R/W	Undefined

Table 5-15   Status Register Field Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
IE	0	Interrupt Enable. Acts as the master enable for software and hardware interrupts: 0: disables interrupts 1: enables interrupts	R/W	Undefined

5.2.12 Cause Register (CP0 Register 13, Select 0)

The *Cause* register primarily describes the cause of the most recent exception. In addition, fields also control software interrupt requests and the vector through which interrupts are dispatched. With the exception of the IP[1:0], IV, and WP fields, all fields in the Cause register are read-only.

Cause Register Format

31	30	29 28 27	24	23	22	21	16	15	10	9 8	7 6 5 4 3 2	1 0
BD	0	CE	0	IV	WP	0		IP[7:2]		IP[1:0]	0	Exc Code	0

Table 5-16   Cause Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	**Bit(s)**			
BD	31	Indicates whether the last exception taken occurred in a branch delay slot: 0: Not in delay slot 1: In delay slot Note that the BD bit is not updated on a new exception if the EXL bit is set.	R	Undefined
CE	29:28	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This field is loaded by hardware on every exception but is unpredictable for all exceptions except for Coprocessor Unusable.	R	Undefined
IV	23	Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector: 0: Use the general exception vector (0x180) 1: Use the special interrupt vector (0x200)	R/W	Undefined

Table 5-16 Cause Register Field Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
WP	22	Indicates that a watch exception was deferred because $Status_{EXL}$ or $Status_{ERL}$ were a one at the time the watch exception was detected. This bit both indicates that the watch exception was deferred and causes the exception to be initiated once $Status_{EXL}$ and $Status_{ERL}$ are both zero. As such, software must clear this bit as part of the watch exception handler to prevent a watch exception loop. Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.	R/W	Undefined
IP[7:2]	15:10	Indicates an external interrupt is pending: 15: Hardware interrupt 5 or timer interrupt 14: Hardware interrupt 4 13: Hardware interrupt 3 12: Hardware interrupt 2 11: Hardware interrupt 1 10: Hardware interrupt 0	R	Undefined
IP[1:0]	9:8	Controls the request for software interrupts: 9: Request software interrupt 1 8: Request software interrupt 0	R/W	Undefined
Exc Code	6:2	Exception code — see Table 5-17.	R	Undefined
0	30, 27:24, 21:16, 7, 1:0	Must be written as zero; returns zero on read.	0	0

Table 5-17 Cause Register ExcCode Field Descriptions

Exception Code Value	Mnemonic	Description
0	Int	Interrupt
1	Mod	TLB modification exception (4Kc core)

Table 5-17   Cause Register ExcCode Field Descriptions (continued)

Exception Code Value	Mnemonic	Description
2	TLBL	TLB exception (load or instruction fetch) (4Kc core)
3	TLBS	TLB exception (store) (4Kc core)
4	AdEL	Address error exception (load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error exception (instruction fetch)
7	DBE	Bus error exception (data reference: load or store)
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
11	CpU	Coprocessor Unusable exception
12	Ov	Integer Overflow exception
13	Tr	Trap exception
14-22	-	Reserved
23	WATCH	Reference to WatchHi/WatchLo address
24	MCheck	Machine check
25-31	-	Reserved

5.2.13 Exception Program Counter (CP0 Register 14, Select 0)

The *Exception Program Counter (EPC)* is a read/write register that contains the address at which processing resumes after an exception has been serviced. All bits of the *EPC* register are significant and must be writable.

For synchronous(precise) exceptions, the *EPC* contains one of the following:

- The virtual address of the instruction that was the direct cause of the exception

- The virtual address of the immediately preceding branch or jump instruction, when the exception causing instruction is in a branch delay slot and the *Branch Delay* bit in the *Cause* register is set.

On new exceptions, the processor does not write to the *EPC* register when the EXL bit in the *Status* register is set. However, the register can still be written via the MTC0 instruction.

EPC Register Format

31 0
EPC

Table 5-18   EPC Register Field Description

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
EPC	31:0	Exception Program Counter.	R/W	Undefined

5.2.14 Processor Identification (CP0 Register 15, Select 0)

The *Processor Identification (PRId)* register is a 32 bit read-only register that contains information identifying the manufacturer, manufacturer options, processor identification, and revision level of the processor.

Processor Identification Register Format

31 24	23 16	15 8	7 0
R	Company ID	Processor ID	Revision

Table 5-19   PRId Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
R	31:24	Reserved. Must be ignored on write and read as zero	R	Preset
Company ID	23:16	Identifies the company that designed or manufactured the processor. In all three cores this field contains a value of 1 to indicate MIPS Technologies, Inc.	R	Preset
Processor ID	15:8	Identifies the type of processor. This field allows software to distinguish between the various types of MIPS Technologies processors. For the 4Kc processor, this field contains a value of 0x80. For the 4Kp and 4Km processors, the value is 0x83.	R	Preset
Revision	7:0	Specifies the revision number of the processor. This field allows software to distinguish between one revision and another of the same processor type.	R	Preset

5.2.15 Config Register (CP0 Register 16, Select 0)

The *Config* register specifies various configuration and capabilities information. Most of the fields in the *Config* register are initialized by hardware during the Reset exception process, or are constant. One field, K0, must be initialized by software in the Reset exception handler.

Config Register Format — Select 0

31	30 28	27 25	24 21	20	19	18 17	16	15	14 13	12 10	9 7	6 3	2 0
M	K23	KU	R	MDU	R	MM	BM	BE	AT	AR	MT	0	K0

Table 5-20   Config Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
M	31	This bit is hardwired to '1' to indicate the presence of the Config1 register.	R	1
K23	30:28	This field controls the cacheability of the kseg2 and kseg3 address segments in BAT implementations. This field is valid in the 4Kp and 4Km processor and is reserved in the 4Kc processor. Refer to Table 5-21 for the field encoding.	BAT: R/W TLB: R	BAT: 010 TLB: 000
KU	27:25	This field controls the cacheability of the kuseg and useg address segments in BAT implementations. This field is valid in the 4Kp and 4Km processor and is reserved in the 4Kc processor. Refer to Table 5-21 for the field encoding.	BAT: R/W TLB: R	BAT: 010 TLB: 000
0	24:21	Must be written as 0. Returns 0 on read.	0	0
MDU	20	This bit indicates the MDU type. 0 = Fast Multiplier Array (4Kc and 4Km cores) 1 = Iterative multiplier (4Kp cores)	R	Preset
0	19	Must be written as 0. Returns 0 on read.	0	0

Table 5-20   Config Register Field Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
MM	18:17	This field contains the merge mode for the 32-byte collapsing write buffer: 00 = No Merging 01 = SysAD Valid merging 10 = Full merging 11 = Reserved	R	Externally Set
BM	16	Burst order. 0: Sequential 1: SubBlock	R	Externally Set
BE	15	Indicates the endian mode in which the processor is running: 　0: Little endian 　1: Big endian	R	Externally Set
AT	14:13	Architecture type implemented by the processor. This field is always 00 to indicate MIPS32.	R	00
AR	12:10	Architecture revision level. This field is always 000 to indicate revision 1. 　0: Revision 1 　1-7: Reserved	R	000
MT	9:7	MMU Type: 1: Standard TLB (4Kc core) 3: Fixed Mapping (4Kp, 4Km cores) 0, 2, 4-7: Reserved	R	Preset
0	6:3	Must be written as zero; returns zero on read.	0	0
K0	2:0	Kseg0 coherency algorithm. Refer to Table 5-21 for the field encoding.	R/W	010

Table 5-21   Cache Coherency Attributes

C(2:0) Value	Cache Coherency Attribute
0, 1, 3*, 4, 5, 6	Cacheable, noncoherent, write-through, no write allocate
2*, 7	Uncached
* These two values are required by the MIPS32 architecture. In the 4K processor cores, all other values are not used. For example, values 0, 1, 4, 5 and 6 are not used and are mapped to 3. The value 7 is not used and is mapped to 2. Note that these values do have meaning in other MIPS Technologies processor implementations. Refer to the MIPS32 specification for more information.	

5.2.16 Config1 Register (CP0 Register 16, Select 1)

The *Config1* register is an adjunct to the Config register and encodes additional capabilities information. All fields in the Config1 register are read-only.

The instruction and data cache configuration parameters include encodings for the number of sets per way, the line size, and the associativity. The total cache size for a cache is therefore:

Associativity * Line Size * Sets Per Way

If the line size is zero, there is no cache implemented.

Config1 Register Format — Select 1

31	30 25	24 22	21 19	18 16	15 13	12 10	9 7	6 4	3	2	1	0
0	MMU Size	IS	IL	IA	DS	DL	DA	0	PC	WR	CA	EP	FP

Table 5-22   Config1 Register Field Descriptions — Select 1

Fields		Description	Read/Write	Reset State
Name	**Bit(s)**			
0	31	This bit is reserved to and must be read or written as zero.	R	Preset
MMU Size	30:25	This field contains the number of entries in the TLB minus one. The field is read as 15 decimal in the 4Kc processor and as 0 decimal in the 4Kp and 4Km processors.	R	Preset
IS	24:22	This field contains the number of instruction cache sets per way. Three options are available in all the 4K cores. All others values are reserved: 0x0: 64 0x1: 128 0x2: 256 0x3 - 0x7: Reserved	R	Preset

Table 5-22　Config1 Register Field Descriptions — Select 1 (continued)

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
IL	21:19	This field contains the instruction cache line size. If an instruction cache is present, it must contain a fixed line size of 16 bytes. 0x0: No Icache present 0x3: 16 bytes 0x1, 0x2, 0x4 - 0x7: Reserved	R	Preset
IA	18:16	This field contains the level of instruction cache associativity. 0x0:　Direct mapped 0x1:　2-way 0x2:　3-way 0x3:　4-way 0x4 - 0x7: Reserved	R	Preset
DS	15:13	This field contains the number of data cache sets per way: 0x0: 64 0x1: 128 0x2: 256 0x3 - 0x7:　Reserved	R	Preset
DL	12:10	This field contains the data cache line size. If a data cache is present, it must contain a line size of 16 bytes. 0x0: No Dcache present 0x3: 16 bytes 0x1, 0x2, 0x4 - 0x7: Reserved	R	Preset
DA	9:7	This field contains the type of set associativity for the data cache: 0x0: Direct mapped 0x1: 2-way 0x2: 3-way 0x3: 4-way 0x4 - 0x7: Reserved	R	Preset

Table 5-22   Config1 Register Field Descriptions — Select 1 (continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
0	6:5	Must be written as zero; returns zero on read.	0	0
PC	4	Performance Counter registers implemented. Always a 0 since the cores do not implement any.	R	0
WR	3	Watch registers implemented. This bit is always read as 1 since the cores each contain one pair of Watch registers.	R	1
CA	2	Code compression (MIPS16™) implemented. This bit is always read as 0 because MIPS16 is not supported.	R	0
EP	1	EJTAG present: This bit is always set to indicate that the core implements EJTAG.	R	1
FP	0	FPU implemented. This bit is always zero since the core does not contain a floating point unit.	R	0

5.2.17 Load Linked Address (CP0 Register 17, Select 0)

The *LLAddr* register contains the physical address read by the most recent Load Linked (LL) instruction. This register is for diagnostic purposes only and serves no function during normal operation.

Load Linked Address Register Format

31 28	27 0
0	PAddr[31:4]

Table 5-23   LLAddr Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
0	31:28	Must be written as zero; returns zero on read.	0	0
PAddr[31:4]	27:0	This field encodes the physical address read by the most recent Load Linked instruction.	R	Undefined

5.2.18 WatchLo Register (CP0 Register 18)

The *WatchLo* and *WatchHi* registers together provide the interface to a watchpoint debug facility that initiates a watch exception if an instruction or data access matches the address specified in the registers. As such, they duplicate some functions of the EJTAG debug solution. Watch exceptions are taken only if the EXL and ERL bits are zero in the *Status* register. If either bit is a one, the WP bit is set in the *Cause* register, and the watch exception is deferred until both the EXL and ERL bits are zero.

The *WatchLo* register specifies the base virtual address and the type of reference (instruction fetch, load, store) to match.

WatchLo Register Format

31		3	2	1	0
VAddr			I	R	W

Table 5-24   WatchLo Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
VAddr	31:3	This field specifies the virtual address to match. Note that this is a doubleword address, since bits [2:0] are used to control the type of match.	R/W	Undefined
I	2	If this bit is set, watch exceptions are enabled for instruction fetches that match the address.	R/W	0 for Cold Reset only.
R	1	If this bit is set, watch exceptions are enabled for loads that match the address.	R/W	0 for Cold Reset only.
W	0	If this bit is set, watch exceptions are enabled for stores that match the address.	R/W	0 for Cold Reset only.

5.2.19 WatchHi Register (CP0 Register 19)

The *WatchLo* and *WatchHi* registers together provide the interface to a watchpoint debug facility that initiates a watch exception if an instruction or data access matches the address specified in the registers. As such, they duplicate some functions of the EJTAG debug solution. Watch exceptions are taken only if the EXL and ERL bits are zero in the *Status* register. If either bit is a one, the WP bit is set in the *Cause* register, and the watch exception is deferred until both the EXL and ERL bits are zero.

The *WatchHi* register contains information that qualifies the virtual address specified in the *WatchLo* register: an ASID, a G(lobal) bit, and an optional address mask. If the G bit is 1, any virtual address reference that matches the specified address will cause a watch exception. If the G bit is a 0, only those virtual address references for which the ASID value in the *WatchHi* register matches the ASID value in the *EntryHi* register cause a watch exception. The optional mask field provides address masking to qualify the address specified in *WatchLo*.

WatchHi Register Format

31	30	29 24	23 16	15 12	11 3	2 1 0
0	G	0	ASID	0	MASK	0

Table 5-25   WatchHi Register Field Descriptions

Name	Bit(s)	Description	Read/Write	Reset State
0	31	Must be written as zero; returns zero on read.	0	0
G	30	If this bit is one, any address that matches that specified in the *WatchLo* register causes a watch exception. If this bit is zero, the ASID field of the *WatchHi* register must match the ASID field of the *EntryHi* register to cause a watch exception.	R/W	Undefined
0	29:24	Must be written as zero; returns zero on read.	0	0
ASID	23:16	ASID value which is required to match that in the *EntryHi* register if the G bit is zero in the *WatchHi* register.	R/W	Undefined
0	15:12	Must be written as zero; returns zero on read.	0	0

Table 5-25   WatchHi Register Field Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
Mask	11:3	Bit mask that qualifies the address in the *WatchLo* register. Any bit in this field that is a set inhibits the corresponding address bit from participating in the address match.	R/W	Undefined
0	2:0	Must be written as zero; returns zero on read.	0	0

5.2.20 Debug Register (CP0 Register 23)

The Debug register is used to control the debug exception and provide information about the cause of the debug exception and when re-entering at the debug exception vector due to a normal exception in debug mode. The read only information bits are updated every time the debug exception is taken or when a normal exception is taken when already in debug mode.

Only the DM bit and the EJTAGver field are valid when read from non-debug mode; the value of all other bits and fields is UNPREDICTABLE. Operation of the processor is UNDEFINED if the Debug register is written from non-debug mode.

Some of the bits and fields are only updated on debug exceptions and/or exceptions in debug mode, as shown below:

- DSS, DBp, DDBL, DDBS, DIB, DINT are updated on both debug exceptions and on exceptions in debug modes

- DExcCode is updated on exceptions in debug mode, and is undefined after a debug exception

- Halt and Doze are updated on a debug exception, and is undefined after an exception in debug mode

- DBD is updated on both debug and on exceptions in debug modes

All bits and fields are undefined when read from normal mode, except those explicitly described to be defined, e.g. EJTAGver and DM.

Debug Register Format

31	30	29	28	27	26	25	24	23 22	21	20	19 18 17	15 14	10	9	8	7 6	5	4	3	2	1	0
DBD	DM	R	LSNM	Doze	Halt	CountDM	IBusEP	R	DBusEP	IEXI	R	Ver	DExcCode	R	SSt	R	DINT	DIB	DDBS	DDBL	DBp	DSS

Table 5-26   Debug Register Field Descriptions

Fields		Description	Read/ Write	Reset
Mnemonic	Bit(s)			
DBD	31	Indicates whether the last debug exception or exception in debug mode, occurred in a branch delay slot: 0: Not in delay slot 1: In delay slot	R	Undefined
DM	30	Indicates that the processor is operating in debug mode: 0: Processor is operating in non-debug mode 1: Processor is operating in debug mode	R	0
R	29	Reserved. Must be written as zero; returns zero on read.	0	0
LSNM	28	Controls access of load/store between dseg and remain memory: 0: Load/stores in dseg address range goes to dseg. 1: Load/stores in dseg address range goes to remain memory.	R/W	0
Doze	27	Indicates that the processor was in any kind of low power mode when a debug exception occurred: 0: Processor not in low power mode when debug exception occurred 1: Processor in low power mode when debug exception occurred	R	Undefined
Halt	26	Indicates that the internal system bus clock was stopped when the debug exception occurred: 0: Internal system bus clock stopped 1: Internal system bus clock running	R	Undefined

Table 5-26   Debug Register Field Descriptions (continued)

Fields		Description	Read/ Write	Reset
Mnemonic	**Bit(s)**			
CountDM	25	Indicates the Count register behavior in debug mode. Encoding of the bit is: 0: Count register stopped in debug mode 1: Count register is running in debug mode	R	1
IBusEP	24	Instruction fetch Bus Error exception Pending. Set when an instruction fetch bus error event occurs or if a 1 is written to the bit by software.Cleared when a Bus Error exception on instruction fetch is taken by the processor, and by reset. If IBusEP is set when IEXI is cleared, a Bus Error exception on instruction fetch is taken by the processor, and IBusEP is cleared.	R/W1	0
R	23:22	Reserved. Must be written as zero; returns zero on read.	0	0
DBusEP	21	Data access Bus Error exception Pending.Covers imprecise bus errors on data access, similar to behavior of IBusEP for imprecise bus errors on an instruction fetch.	R/W1	0
IEXI	20	Imprecise Error eXception Inhibit controls exceptions taken due to imprecise error indications. Set when the processor takes a debug exception or exception in debug mode. Cleared by execution of the DERET instruction. Other-wise modifiable by debug mode software. When IEXI is set then the imprecise error exceptions from bus error on instruction fetch or data access, cache error or machine check are inhibited and deferred until the bit is cleared.	R/W	0
R	19:18	Reserved. Must be written as zero; returns zero on read.	0	0

Table 5-26   Debug Register Field Descriptions (continued)

Fields		Description	Read/ Write	Reset
Mnemonic	**Bit(s)**			
Ver	17:15	EJTAG version	R	1
DExcCode	14:10	Indicates the cause of the latest exception in debug mode. The field is encoded as the Exc-Code field in the Cause register for those normal exceptions that may occur in debug mode. Value is undefined after a debug exception.	R	Undefined
R	9	Reserved. Must be written as zero; returns zero on read.	0	0
SSt	8	Controls if debug single step exception is enabled: 0: No debug single step exception enabled 1: Debug single step exception enabled	R/W	0
R	7:6	Reserved. Must be written as zero; returns zero on read.	0	0
DINT	5	Indicates that a debug interrupt exception occurred. Cleared on exception in debug mode. 0: No debug interrupt exception 1: Debug interrupt exception	R	Undefined
DIB	4	Indicates that a debug instruction break exception occurred. Cleared on exception in debug mode. 0: No debug instruction exception 1: Debug instruction exception	R	Undefined
DDBS	3	Indicates that a debug data break exception occurred on a store. Cleared on exception in debug mode. 0: No debug data exception on a store 1: Debug instruction exception on a store	R	Undefined

Table 5-26   Debug Register Field Descriptions (continued)

Fields		Description	Read/ Write	Reset
Mnemonic	**Bit(s)**			
DDBL	2	Indicates that a debug data break exception occurred on a load. Cleared on exception in debug mode. 0: No debug data exception on a load 1: Debug instruction exception on a load	R	Undefined
DBp	1	Indicates that a debug software breakpoint exception occurred. Cleared on exception in debug mode. 0: No debug software breakpoint exception 1: Debug software breakpoint exception	R	Undefined
DSS	0	Indicates that a debug single step exception occurred. Cleared on exception in debug mode. 0: No debug single step exception 1: Debug single step exception	R	Undefined

5.2.21 Debug Exception Program Counter Register (CP0 Register 24)

The Debug Exception Program Counter (DEPC) register is a read/write register that contains the address at which processing resumes after a debug exception or debug mode exception has been serviced.

For synchronous (precise) debug and debug mode exceptions, the DEPC contains either:

- The virtual address of the instruction that was the direct cause of the debug exception, or

- The virtual address of the immediately preceding branch or jump instruction, when the debug exception causing instruction is in a branch delay slot, and the Debug Branch Delay (BDB) bit in the Debug register is set.

For asynchronous debug exceptions (debug interrupt), the DEPC contains the virtual address of the instruction where execution should resume after the debug handler code is executed.

Debug Exception Program Counter Register Format

31	0
DEPC	

Table 5-27   Debug Register Formats

Fields		Description	Read/ Write	Reset
Mnemonic	Bit(s)			
DEPC	31:0	The DEPC register is updated with the virtual address of the instruction that caused the debug exception. If the instruction is in the branch delay slot, the virtual address of the immediately preceding branch or jump instruction is placed in this register. Execution of the DERET instruction causes a jump to the address in the DEPC.	R/W	Undefined

5.2.22 TagLo Register (CP0 Register 28, Select 0)

The *TagLo* register acts as the interface to the cache tag array. The Index Store Tag and Index Load Tag operations of the CACHE instruction use the *TagLo* register as the source of tag information, respectively. Note that the 4K cores do not implement the TagHi register.

TagLo Register Format

31	10	9	8	7	6	5	4	3	2	1	0
PA		R		Valid				R	L	LRF	R

Table 5-28   TagLo Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
PA	31:10	This field contains the physical address of the cache line being stored.	R/W	Undefined
R	9:8	Must be written as zero; returns zero on read.	0	0
Valid	7:4	This field indicates whether the corresponding word in the cache line is valid in the cache.	R/W	Undefined
R	3	Must be written as zero; returns zero on read.	0	0
L	2	Specifies the lock bit for the cache tag. When this bit is set, the corresponding cache line should not be replaced by the cache replacement algorithm.	R/W	Undefined
LRF	1	LRF. One bit of the LRF bits for the set this cache line is a part of. This bit is inverted every time a new cache line is filled in the cache entry.	R/W	Undefined
R	0	Must be written as zero; returns zero on read.	0	0

5.2.23 DataLo Register (CP0 Register 28, Select 1)

The *DataLo* register is a read-only register that acts as the interface to the cache data array and are intended for diagnostic operations only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *DataLo* register. Note that the 4K cores do not implement the DataHi register.

DataLo Register Format

31	0
DATA	

Table 5-29   DataLo Register Field Description

Fields		Description	Read/W rite	Reset State
Name	Bit(s)			
DATA	31:0	Low-order data read from the cache data array.	R	Undefined

5.2.24 ErrorEPC (CP0 Register 30, Select 0)

The *ErrorEPC* register is a read-write register, similar to the *EPC* register, except that *ErrorEPC* is used on error exceptions. All bits of the *ErrorEPC* register are significant and must be writable. It is also used to store the program counter on Reset, Soft Reset, and nonmaskable interrupt (NMI) exceptions.

The *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. This address can be:

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

Unlike the *EPC* register, there is no corresponding branch delay slot indication for the *ErrorEPC* register.

ErrorEPC Register Format

31	0
ErrorEPC	

Table 5-30 ErrorEPC Register Field Description

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
ErrorEPC	31:0	Error Exception Program Counter	R/W	Undefined

5.2.25 DeSave Register (CP0 Register 31)

The Debug Exception Save (DeSave) register is a read/write register that functions as a simple memory location. This register is used by the debug exception handler to save one of the GPRs that is then used to save the rest of the context to a pre-determined memory area (such as in the EJTAG Probe). This register allows the safe debugging of exception handlers and other types of code where the existence of a valid stack for context saving cannot be assumed.

DeSave Register Format

31	0
DESAVE	

Table 5-31   DeSave Register Description

Bit(s)	Mnemonic	Description	R/W	Reset
31:0	DESAVE	Debug exception save contents.	R/W	Undefined

Hardware and Software Initialization

The MIPS32 4K™ processor cores have only a minimal amount of hardware initialization and rely on software to fully initialize the device.

This chapter contains the following sections:

- Section 6.1, "Hardware Initialized Processor State"
- Section 6.2, "Software Initialized Processor State"

6.1 Hardware Initialized Processor State

The 4K processor cores, like most MIPS processors, are not fully initialized by reset. Only a minimal subset of the processor state is cleared. This is enough to bring the core up while running in unmapped and uncached code space. All other processor states can then be initialized by software. Reset is asserted after power-up to bring the device into a known state. SoftReset can be used when the device is already up and running and does not need as much initialization.

6.1.1   Coprocessor Zero State

Much of the hardware initialization occurs in Coprocessor Zero.

- Random (4Kc core only)- Set to maximum value on Reset

- Wired (4Kc core only)- Set to 0 on Reset

- Status$_{BEV}$ - set to 1 on Reset/SoftReset

- Status$_{TS}$ - cleared to 0 on Reset/SoftReset

- Status$_{SR}$ - cleared to 0 on Reset, set to 1 on SoftReset

- Status$_{NMI}$ - cleared to 0 on Reset/SoftReset

- Status$_{ERL}$ - set to 1 on Reset/SoftReset

- Status$_{RP}$ - set to 0 on Reset

- WatchLo$_{I,R,W}$ - cleared to 0 on Reset

- Config fields related to static inputs - set to input value by Reset

- Config$_{K0}$ - set to 010 on Reset

- Config$_{KU}$ - set to 010 on Reset (4Km™ and 4Kp™ cores only)

- Config$_{K23}$ - set to 010 on Reset (4Km and 4Kp cores only)

- Debug$_{DM}$ - cleared to 0 on Reset/SoftReset (unless EJTAGBOOT option is used to boot into DebugMode, see EJTAG chapter for details)

- Debug$_{LSNM}$ - cleared to 0 on Reset/SoftReset

- Debug$_{IBusEP}$ - cleared to 0 on Reset/SoftReset

- Debug$_{DBusEP}$ - cleared to 0 on Reset/SoftReset

- Debug$_{IEXI}$ - cleared to 0 on Reset/SoftReset

- Debug$_{SSt}$ - cleared to 0 on Reset/SoftReset

6.1.2 TLB Initialization (4Kc core only)

Each 4Kc TLB entry has a "hidden" state bit which is set by Reset/SoftReset and is cleared when the TLB entry is written. This bit disables matches and prevents "TLB Shutdown" conditions from being generated by the power-up values in the TLB array (when two or more TLB entries match on a single address). This bit is not visible to software.

6.1.3 Bus State Machines

All pending bus transactions are aborted and the state machines in the bus interface unit are reset when a Reset or SoftReset exception is taken.

6.1.4 Static Configuration Inputs

All static configuration inputs (defining the bus mode and cache size for example) should only be changed during Reset.

6.1.5 Fetch Address

Upon Reset/SoftReset, unless the EJTAGBOOT option is used, the fetch is directed to VA 0xBFC00000 (PA 0x1FC00000). This address is in KSeg1,which is unmapped and uncached, so that the TLB and caches do not require hardware unitization.

6.2 Software Initialized Processor State

Software is required to initialize the following parts of the device.

6.2.1 Register File

The register file powers up in an unknown state with the exception of r0 which is always 0. Initializing the rest of the register file is not required for proper operation. Good code will generally not read a register before writing to it, but the boot code can initialize the register file for added safety.

6.2.2   TLB (4Kc Core Only)

Because of the hidden bit indicating initialization, the 4Kc core does not require TLB initialization upon ColdReset. This is an implementation specific feature of the 4Kc core and cannot be relied upon if writing generic code for MIPS32/64 processors. When initializing the TLB, care must be taken to avoid creating a "TLB Shutdown" condition where two TLB entries could match on a single address. Unique virtual addresses should be written to each TLB entry to avoid this

6.2.3   Caches

The cache tag and data arrays power up to an unknown state and are not affected by reset. Every tag in the cache arrays should be initialized to an invalid state using the CACHE instruction (typically the Index Invalidate function). This can be a long process, especially since the instruction cache initialization needs to be run in an uncached address region.

6.2.4   Coprocessor Zero state

Miscellaneous Cop0 state needs to be initialized prior to leaving the boot code. There are various exceptions which are blocked by ERL=1 or EXL=1 and which are not cleared by Reset. These can be cleared to avoid taking spurious exceptions when leaving the boot code.

- Cause: WP (Watch Pending), SW0/1 (Software Interrupts) should be cleared.

- Config: K0 should be set to the desired Cache Coherency Algorithm (CCA) prior to accessing KSeg0.

- Config: (4Km and 4Kp cores only) KU and K23 should be set to the desired CCA for USeg/KUSeg and KSeg2/3 respectively prior to accessing those regions.

- Count: Should be set to a known value if Timer Interrupts are used.

- Compare: Should be set to a known value if Timer Interrupts are used. The write to compare will also clear any pending Timer Interrupts (Thus, Count should be set before Compare to avoid any unexpected interrupts).

- Status: Desired state of the device should be set.

- Other Cop0 state: Other registers should be written before they are read. Some registers are not explicitly writeable, and are only updated as a by-product of instruction execution or a taken exception. Uninitialized bits should be masked off after reading these registers.

Caches

The instruction and data cache controllers of the MIPS32 4K™ processor cores support caches of various sizes, organizations, and set-associativity. For example, the data cache can be 2 Kbytes in size and 2-way set associative, while the instruction cache can be 8 Kbytes in size and 4-way set associative. Each cache can each be accessed in a single processor cycle. In addition, each cache has its own 32-bit data path and both caches can be accessed in the same pipeline clock cycle.

This chapter contains the following sections.

- Section 7.1, "Cache Protocols"

- Section 7.2, "Instruction Cache"

- Section 7.3, "Data Cache"

Table 7-1 lists the instruction and data cache attributes:

Table 7-1   Instruction and Data Cache Attributes

Parameter	Instruction	Data
Size	0 - 16 Kbytes	0 - 16 Kbytes
Number of Cache Sets	0, 64, 128 and 256	0, 64, 128 and 256
Lines Per Set (Associativity)	1 - 4 way set associative	1 - 4 way set associative
Line Size	16 bytes	16 bytes
Read Unit	32-bits	32-bits
Write Policy	N/A	write-through *without write-allocate*
Miss restart after transfer of	miss word	miss word
Cache Locking	per line	per line

The core provides a flexible cache configuration structure that allows the instruction and data caches to be configured in any combination of ways based on the following sizes. All of the cache sizes listed below can be 1-, 2-, 3-, or 4-way set associative.

Table 7-2   Instruction and Data Cache Sizes

Cache Size	Way Organization Options
0K	No cache
1K	One 1K way
2K	One 2K way Two 1K ways
3K	Three 1K ways
4K	One 4K way Two 2K ways Four 1K ways
6K	Three 2K ways
8K	Two 4K ways Four 2K ways
12K	Three 4K ways
16K	Four 4K ways

7.1 Cache Protocols

All the 4K cores support the following cache protocols:

- **Uncached**: Addresses in a memory area indicated as uncached are not read from the cache. Stores to such addresses are written directly to main memory, without changing cache contents.

- **Write-through**: Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is cache resident. If it is resident, the cache contents are updated, and main memory is also written. If the cache lookup misses, only main memory is written.

7.2 Instruction Cache

The instruction cache is an optional on-chip memory block of up to 16 Kbytes. The virtually indexed, physically tagged cache allows the virtual-to-physical address translation to occur in parallel with the cache access rather than having to wait for the physical address translation. The tag contains 22 bits of physical address, 4 valid bits, a lock bit, and the FIFO replacement bit.

All the cores support instruction cache-locking. Cache locking allows critical code or data segments to be locked into the cache on a "per-line" basis, enabling the system programmer to maximize the efficiency of the system cache.

The cache locking function is always enabled on all instruction cache entries. Entries can then be marked as locked or unlocked on a per entry basis using the CACHE instruction.

7.3 Data Cache

The data cache is an optional on-chip memory block of up to 16 Kbytes. The virtually indexed, physically tagged cache allows the virtual-to-physical address translation to occur in parallel with the cache access rather than having to wait for the physical address translation. The tag contains 22 bits of physical address, 4 valid bits, a lock bit, and the FIFO replacement bit.

In addition to instruction cache locking, the core also supports a data cache locking mechanism identical to the instruction cache. Critical data segments to be locked into the cache on a "per-line" basis. The locked contents can be updated on a store hit, but cannot be selected for replacement on a store miss.

The cache locking function is always enabled on all data cache entries. Entries can then be marked as locked or unlocked on a per entry basis using the CACHE instruction.

Power Management

The MIPS32 4K™ processor cores offer a number of power management features, including low-power design, active power management and power-down modes of operation. The core is a static design that supports a WAIT instruction designed to signal the rest of the device that execution and clocking should be halted, reducing system power consumption during idle periods.

The core provides two mechanisms for system level low power support discussed in the following sections.

- Section 8.1, "Register Controlled Power Management"
- Section 8.2, "Instruction Controlled Power Management"

8.1 Register Controlled Power Management

The RP bit in the CP0 Status register a standard software mechanism for placing the system into a low power state. The state of the RP bit is available externally via the SI_RP signal. Three additional pins, SI_EXL, SI_ERL, and EJ_DebugM support the power management function by allowing the user to change the power state if an exception or error occurs while the core is in a low power state.

Setting the RP bit of the CP0 Status register causes the core to assert the SI_RP signal. The external agent can then decide whether to reduce the clock frequency and place the core into power down mode.

If an interrupt is taken while the device is in power down mode, that interrupt may need to be serviced depending on the needs of the application. The interrupt causes an exception which in turn causes the EXL bit to be set. The setting of the EXL bit causes the assertion of the SI_EXL signal on the external bus, indicating to the external agent that an interrupt has occurred. At this time the external agent can choose to either speed up the clocks and service the interrupt or let it be serviced at the lower clock speed.

The setting of the ERL bit causes the assertion of the SI_ERL signal on the external bus, indicating to the external agent that an error has occurred. At this time the external agent can choose to either speed up the clocks and service the error or let it be serviced at the lower clock speed.

Similarly, the EJ_DebugM signal indicates that the processor is in debug mode. Debug mode is entered when the processor takes a debug exception. If fast handling of this is desired, the external agent can speed up the clocks.

The core provides 4power down signals that are part of the system interface. Three of the pins change state as the corresponding bits in the CP0 *Status* register are set or cleared. The fourth pin indicates that the processor is in debug mode.

- The SI_RP signal represents the state of the RP bit (27) in the CP0 Status register.

- The SI_EXL signal represents the state of the EXL bit (1) in the CP0 Status register.

- The SI_ERL signal represents the state of the ERL bit (2) in the CP0 Status register.

- The EJ_DebugM signal indicates that the processor has entered debug mode

8.2 Instruction Controlled Power Management

The second mechanism for invoking power down mode is through execution of the WAIT instruction. If the bus is idle at the time the WAIT instruction reaches the M stage of the pipeline the internal clocks are suspended and the pipeline is frozen. However, the internal timer and some of the input pins (SI_Int[5:0], SI_NMI, SI_Reset, SI_ColdReset, and EJ_DINT) continue to run. If the bus is not idle at the time the WAIT instruction reaches the

M stage, the pipeline stalls until the bus becomes idle, at which time the clocks are stopped. Once the CPU is in instruction controlled power management mode, any enabled interrupt, NMI, debug interrupt through EJ_DINT, or reset condition causes the CPU to exit this mode and resume normal operation. While the part is in this low-power mode, the SI_SLEEP signal is asserted to indicate to external agents what the state of the chip is.

EJTAG Debug Support

The EJTAG debug logic in the MIPS32 4K™ processor cores provide two optional modules, one for hardware breakpoints, and the other a Test Access Port (TAP) for a dedicated connection to a debug host.

This chapter contains the following sections.

- Section 9.1, "Debug Control Register"
- Section 9.2, "Hardware Breakpoints"
- Section 9.3, "Test Access Port Operation"
- Section 9.4, "Test Access Port (TAP) Instructions"
- Section 9.5, "EJTAG Registers"
- Section 9.6, "Processor Accesses"

9.1 Debug Control Register

The Debug Control Register (DCR) register controls and provides information about debug issues, and is always provided with the CPU core. The register is memory mapped in drseg at offset 0x0.

The DataBrk and InstBrk bits indicates if hardware breakpoints are included in the implementation, and debug software is expected to read hardware breakpoint registers for additional information.

Hardware and software interrupts are maskable for non-debug mode with the INTE bit, which works in addition to the other mechanisms for interrupt masking and enabling. NMI is maskable in non-debug mode with the NMIE bit, and a pending NMI is indicated through the NMIP bit.

The SRE bit allows implementation dependent masking of none, some or all sources for soft reset. The soft reset masking may only be applied to a soft reset source, if that source can be efficiently masked in the system, thus resulting on no reset at all. If that is not possible, then that soft reset source should not be masked, since a "half" soft reset may cause the system to fail or hang. There is no automatic indication of whether the SRE is effective, but the user must consult system documentation.

The PE bit reflects the ProbEn bit from the EJTAG Control register (ECR), whereby the probe can indicate to the debug software running on the CPU if the probe expects to service dmseg accesses. The reset value in the table below takes effect on both hard and soft reset.

Debug Control Register

31 30	29	28 18	17	16	15 5	4	3	2	1	0
Res	ENM	Res	DB	IB	Res	INTE	NMIE	NMIP	SRE	PE

Table 9-1   Debug Control Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
Res	31:30	reserved	R	0
ENM	29	**Endianess in Kernel and Debug mode** This bit indicates the endianess in Kernel and Debug mode. 0: Little Endian 1: Big Endian	R	Preset
Res	28:18	reserved	R	0

Table 9-1   Debug Control Register Field Descriptions (continued)

| Fields | | Description | Read/Write | Reset State |
Name	Bit(s)			
DB	17	**Data Break Implemented** This bit indicates if the Data Break feature is implemented. 0: No Data Break feature implemented 1: Data Break feature is implemented	R	Preset
IB	16	**Instruction Break Implemented** This bit indicates if the Instruction Break feature is implemented. 0: No Instruction Break feature implemented 1: Instruction Break feature is implemented	R	Preset
Res	5:15	reserved	R	0
INTE	4	Interrupt Enable in Normal Mode. This bit provides the hardware and software interrupt enable for non-debug mode, in addition to other masking mechanisms: 0: Interrupt disabled. 1: Interrupts enabled (depending on other enabling mechanisms).	R/W	1
NMIE	3	Non-Maskable Interrupt Enable for non-debug mode 0: NMI disabled. 1: NMI enabled.	R/W	1
NMIP	2	NMI Pending Indication. 0: No NMI pending. 1: NMI pending.	R	0
SRE	1	Soft Reset Enable This bit allows the system to mask soft resets. The core does not internally mask soft reset. Rather the state of this bit appears on the EJ_SRstE external output signal, allowing the system to mask soft resets if desired.	R/W	1

Table 9-1   Debug Control Register Field Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
PE	0	Probe Enable This bit reflects the ProbEn bit in the EJTAG Control register. 0: No accesses to dmseg allowed 1: EJTAG probe services accesses to dmseg	R	Same value as ProbEn in ECR (see Table 9-23)

9.2 Hardware Breakpoints

Hardware breakpoints provide for the comparison by hardware of executed instructions and data load/store transactions. It is possible to set instruction breakpoints on addresses even in ROM area, and set data breakpoints to cause a debug exception on a specific data transaction. Instruction and data hardware breakpoints are alike for may aspects, and are thus described in parallel in the following. The term hardware is not applied to breakpoint, unless required to distinguish it from software breakpoint.

There are two types of simple hardware breakpoints implemented in the 4K cores; Instruction breakpoints and Data breakpoints.

Each core can be configured with the following breakpoint options:

* No data or instruction breakpoints

* Two instruction and one data breakpoint

* Four instruction and two data breakpoints

9.2.1   Features of Instruction Breakpoint

Instruction breaks occur on instruction fetch operations and the break is set on virtual address on the bus between the CPU and the instruction cache. Instruction breaks can also be made on the ASID value used by the MMU. Finally, a mask can be applied to the virtual address to set breakpoints on a range of instructions.

Instruction breakpoints compare the virtual address of the executed instructions (PC) and the ASID, with the registers for each instruction breakpoint including masking of address and ASID. Overview is shown in Figure 9-1.
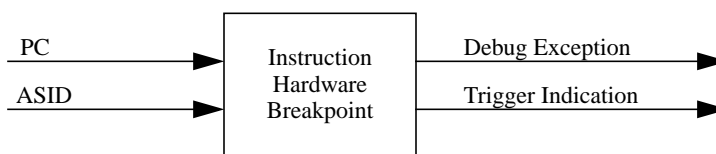
Figure 9-1   Instruction Hardware Breakpoint Overview

When a instruction breakpoint matches, a debug exception and/or a trigger is generated. An internal bit in the instruction breakpoint registers is set to indicate that the match occurred.

9.2.2   Features of Data Breakpoint

Data breakpoints occur on load/store transactions. Breakpoints are set on virtual address and ASID values, similar to the Instruction breakpoint. Data breakpoints can be set on a load, a store or both. Data breakpoints can also be set based on the value of the load/store operation. Finally, masks can be applied to both the virtual address and the load/store value.

Data breakpoints compare the transaction type (TYPE), which may be load or store, the virtual address of the transaction (ADDR), the ASID, accessed bytes (BYTELANE) and data value (DATA), with the registers for each data breakpoint including masking or qualification on the transaction properties. An overview is shown in Figure 9-2.
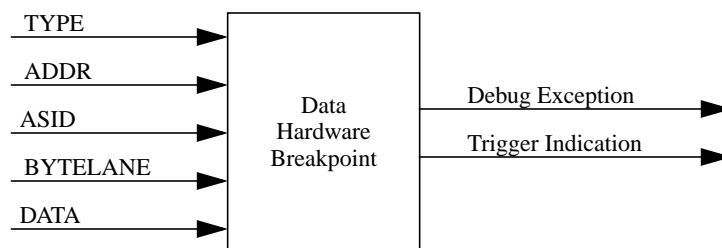
Figure 9-2   Data Hardware Breakpoint Overview

When a data breakpoint matches, a debug exception and/or a trigger is generated, and an internal bit in the data breakpoint registers is set to indicate that the match occurred. The match is either precise whereby the debug exception or trigger occurs on the instruction that caused the breakpoint to match, or it is imprecise whereby the debug exception or trigger occurs later in the program flow.

9.2.3   Overview of Registers for Instruction Breakpoint

The register with implementation indication and status for instruction breakpoints in general is shown in Table 9-2.

Table 9-2   Overview of Status Register for Instruction Breakpoints

Register Mnemonic	Register Name and Description
IBS	Instruction Breakpoint Status

The four instruction breakpoints are numbered 0 to 3 for registers and breakpoints, and the number is indicated by n. The registers for each breakpoint are shown in Figure 9-3

Table 9-3   Overview of Registers for each Instruction Breakpoint

Register Mnemonic	Register Name and Description
IBAn	Instruction Breakpoint Address n
IBMn	Instruction Breakpoint Address Mask n
IBASIDn	Instruction Breakpoint ASID n
IBCn	Instruction Breakpoint Control n

9.2.4   Registers for Data Breakpoint Setup

The register with implementation indication and status for data breakpoints in general is shown in Table 9-4.

Table 9-4   Overview of Status Register for Data Breakpoints

Register Mnemonic	Register Name and Description
DBS	Data Breakpoint Status

The two data breakpoints are numbered 0 and 1 for registers and breakpoints, and the number is indicated by n. The registers for each breakpoint are shown in Table 9-5.

Table 9-5   Overview of Registers for each Data Breakpoint

Register Mnemonic	Register Name and Description
DBAn	Data Breakpoint Address n

Table 9-5   Overview of Registers for each Data Breakpoint

Register Mnemonic	Register Name and Description
DBMn	Data Breakpoint Address Mask n
DBASIDn	Data Breakpoint ASID n
DBCn	Data Breakpoint Control n
DBVn	Data Breakpoint Value n

9.2.5   Conditions for Matching Breakpoints

A number of conditions must be fulfilled in order for a breakpoint to match on an executed instruction or a data transaction, and the conditions for matching instruction and data breakpoints are described below. The breakpoints only matches for instructions executed in non-debug mode, thus never on instructions executed in debug mode.

The match of an enabled breakpoint can either generate a debug exception or a trigger indication. The BE and/or TE bits in the IBCn or DBCn registers are used to enable the breakpoints.

Debug software should not configure breakpoints to compare on ASID value, unless a TLB is present in the implementation.

9.2.5.1 Conditions for Matching Instruction Breakpoint

When an instruction breakpoint is enabled, that breakpoint is evaluated for the address of every executed instruction in non-debug mode, including execution of instructions at an address causing an address error on instruction fetch. The breakpoint is not evaluated on instructions from speculative fetch or execution, nor for addresses which are unaligned with an executed instruction.

Match of the breakpoint depends on the virtual address of the executed instruction (PC) which can be masked at bit level, and match may also include optional compare of ASID value. The registers for each instruction breakpoint has the values and mask used in the compare, and the equation that determines the match is shown below in C-like notation.

```
IB_match
            ( ! IBCn_ASIDuse || ( ASID == IBASIDn_ASID ) ) &&
            ( <all 1's> == ( IBMn_IBM | ~ ( PC ^ IBAn_IBA ) ) )
```

The match indication for data breakpoints is always precise, i.e. indicated on the instruction causing the IB_match to be true.

9.2.5.2 Conditions for Matching Data Breakpoints

When a data breakpoint is enabled, that breakpoint is evaluated for every data transaction due to a load/store instruction executed in non-debug mode, including load/store for coprocessor, and transactions causing an address error on data access. The breakpoint is not evaluated due to PREF instruction or other transactions which are not part of explicit load/store transactions in the execution flow, nor for addresses which are not the explicit load/store source or destination address.

Match of the breakpoint depends on the transaction type (TYPE) as load or store, the address, and optionally the data value of a transaction. The registers for each data breakpoint has the values and mask used in the compare, and the equations that determines the match is shown below in C-like notation.

The overall match equation is the DB_match.

```
DB_match =
          ( ( ( TYPE == load ) && ! DBCn_NoLB ) || ( ( TYPE == store ) && ! DBCn_NoSB ) ) &&
          DB_addr_match && ( DB_no_value_compare || DB_value_match )
```

Match on the address part, DB_addr_match, depends on virtual address of the transaction (ADDR), the ASID value, and the accessed bytes (BYTELANE) where BYTELANE[0] is 1 only if the byte at bits [7:0] on the bus is accessed, and BYTELANE[1] is 1 only if byte at bits [15:8] is accessed, etc. The DB_addr_match is shown below.

```
DB_addr_match =
          ( ! DBCn_ASIDuse || ( ASID == DBASIDn_ASID ) ) &&
          ( <all 1's> == ( DBMn_DBM | ~ ( ADDR ^ DBAn_DBA ) ) ) &&
          ( <all 0's> != ( ~ BAI & BYTELANE ) )
```

The size of $DBCn_{BAI}$ and BYTELANE is 4 bits.

Data value compare is included in the match condition for the data breakpoint depending on the bytes (BYTELANE as described above) accessed by the transaction, and the contents of breakpoint registers. The DB_no_value_compare is shown below.

```
DB_no_value_compare =
          ( <all 1's> == ( DBCn_BLM | DBCn_BAI | ~ BYTELANE ) )
```

The size of $DBCn_{BLM}$, $DBCn_{BAI}$ and BYTELANE is 4 bits.

In case data value compare is required, DB_no_value_compare is false, then the data value from the data bus (DATA) is compared and masked with the registers for the data breakpoint. The endianess is not considered in these match equations for value, as the compare uses the data bus value directly, thus debug software is responsible for setup of the breakpoint corresponding with endianess.

```
DB_value_match =
```

$$((\text{DATA}[7:0] == \text{DBVn}_{DBV[7:0]}) \ || \ ! \ \text{BYTELANE}[0] \ || \ \text{DBCn}_{BLM[0]} \ || \ \text{DBCn}_{BAI[0]}) \ \&\&$$
$$((\text{DATA}[15:8] == \text{DBVn}_{DBV[15:8]}) \ || \ ! \ \text{BYTELANE}[1] \ || \ \text{DBCn}_{BLM[1]} \ || \ \text{DBCn}_{BAI[1]}) \ \&\&$$
$$((\text{DATA}[23:16] == \text{DBVn}_{DBV[23:16]}) \ || \ ! \ \text{BYTELANE}[2] \ || \ \text{DBCn}_{BLM[2]} \ || \ \text{DBCn}_{BAI[2]}) \ \&\&$$
$$((\text{DATA}[31:24] == \text{DBVn}_{DBV[31:24]}) \ || \ ! \ \text{BYTELANE}[3] \ || \ \text{DBCn}_{BLM[3]} \ || \ \text{DBCn}_{BAI[3]})$$

The match for a data breakpoint is always precise, since the match expression is fully evaluated at the time the load/store instruction is executed. A true DB_match can thereby be indicated on the very same instruction causing the DB_match to be true.

9.2.6 Debug Exceptions from Breakpoints

Instruction and data breakpoints may be setup to generate a debug exception when the match condition is true, as described below.

9.2.6.1 Debug Exception by Instruction Breakpoint

If the breakpoint is enabled by BE in the IBCn register, then a debug instruction break exception occurs if the IB_match equation is true. The corresponding BS[n] bit in the IBS register is set when the breakpoint generates the debug exception.

The debug instruction break exception is always precise, so the DEPC register and DBD bit in the Debug register points to the instruction that caused the IB_match equation to be true.

The instruction receiving the debug exception does not update any registers due to the instruction, nor does any load or store by that instruction occur. Thus a debug exception from a data breakpoint can not occur for instructions receiving a debug instruction break exception.

The debug handler usually returns to the instruction causing the debug instruction break exception, whereby the instruction is executed. Debug software is responsible for disabling the breakpoint when returning to the instruction, otherwise the debug instruction break exception reoccurs.

9.2.6.2 Debug Exception by Data Breakpoint

If the breakpoint is enabled by BE in the DBCn register, then a debug exception occurs when the DB_match condition is true. The corresponding BS[n] bit in the DBS register is set when the breakpoint generates the debug exception.

A debug data break exception occurs when a data breakpoint indicates a match. In this case the DEPC register and DBD bit in the Debug register points to the instruction that caused the DB_match equation to be true.

The instruction causing the debug data break exception does not update any registers due to the instruction, and the following applies to the load or store transaction causing the debug exception:

- A store transaction is not allowed to complete the store to the memory system.

- A load transaction with no data value compare, i.e. where the DB_no_value_compare is true for the match, is not allowed to complete the load.

- A load transaction for a breakpoint with data value compare must occur from the memory system, since the value is required in order to evaluate the breakpoint.

The result of this is that the load or store instruction causing the debug data break exception appears as not executed, with the exception that a load from the memory system do occur for a breakpoint with data value compare, but the result of this load is discarded since the register file is not updated by the load.

If both data breakpoints without and with data value compare would match the same transaction and generate a debug exception, then the following rules apply with respect to updating the BS[n] bits.

- On both a load and store the BS[n] bits are required to be set for all matching breakpoints without data value compare.

- On a store then BS[n] bits are allowed but not required to be set for all matching breakpoints with data value compare, but either all or none of the BS[n] bits must be set for these breakpoints.

- On a load then no of the BS[n] bits are allowed to be set, since the load is not allowed to occur due to the debug exception from a breakpoint without data value compare, and a valid data value is therefore not returned.

Any BS[n] bit set prior to the match and debug exception are kept set, since BS[n] bits are only cleared by debug software.

The debug handler usually returns to the instruction causing the debug data break exception, whereby the instruction is re-executed. This re-execution may result in a repeated load from system memory, since the load may have occurred previously in order to evaluate the breakpoint as described above. I/O devices with side effects on load must be able to allow such reloads, or debug software should alternatively avoid setting data breakpoint with data value compare on such I/O devices. Debug software is responsible for disabling breakpoints when returning to the instruction, otherwise the debug data break exception will reoccur.

9.2.7   Breakpoint used as Triggerpoint

Both instruction and data hardware breakpoints may be setup by software so a matching breakpoint does not generate a debug exception, but only an indications through the BS[n] bit. The TE bit in the IBCn or DBCn register

controls if a instruction respectively data breakpoint is used as a so-called triggerpoint. The triggerpoints are, like breakpoints, only compared for instructions executed in non-debug mode.

The BS[n] bit in the IBS or DBS register is set when the respective IB_match or DB_match bit is true.

9.2.8   Instruction Breakpoint Registers

The registers for instruction breakpoints are described below. These registers have implementation information and are used for setup the instruction breakpoints. All registers are in drseg, and the addresses are shown in section Table 9-6.

Table 9-6   Addresses for Instruction Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x1000	IBS	Instruction Breakpoint Status
0x1100 + n * 0x100	IBAn	Instruction Breakpoint Address n
0x1108 + n * 0x100	IBMn	Instruction Breakpoint Address Mask n
0x1110 + n * 0x100	IBASIDn	Instruction Breakpoint ASID n
0x1118 + n * 0x100	IBCn	Instruction Breakpoint Control n
n is breakpoint number in range 0 to 3		

An example of some of the registers; IBA0 is at offset 0x1100 and IBC2 is at offset 0x1318.

9.2.8.1 Instruction Breakpoint Status (IBS) Register

Compliance Level: Implemented only if any instruction breakpoints.

The Instruction Breakpoint Status (IBS) register holds implementation and status information about the instruction breakpoints.

The ASIDsup applies to all the instruction breakpoints.

IBS Register Format

31	30	29 28	27 24	23 4	3 0
Res	ASID	Res	BCN	Res	BS

Table 9-7   IBS Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
Res	31	Must be written as zero; returns zero on read.	0	0
ASID	30	Indicates that ASID compare is supported in instruction breakpoints.	R	4Kc core - 1 4Km/ 4Kp cores - 0
Res	29:28	Must be written as zero; returns zero on read.	0	0
BCN	27:24	Number of instruction breakpoints implemented	R	4
Res	23:4	Must be written as zero; returns zero on read.	0	0
BS	3:0	Break status for breakpoint n is at BS[n], with n as 0 to 3. The bit is set to 1 when the condition for the corresponding breakpoint has matched.	R/W	Undefined

9.2.8.2 Instruction Breakpoint Address n (IBAn) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

The Instruction Breakpoint Address n (IBAn) register has the address used in the condition for instruction breakpoint n

IBAn Register Format

31	0
IBA	

Table 9-8   IBAn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
IBA	31:0	Instruction breakpoint address for condition	R/W	Undefined

9.2.8.3 Instruction Breakpoint Address Mask n (IBMn) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

The Instruction Breakpoint Address Mask n (IBMn) register has the mask for address compare used in the condition for instruction breakpoint n.

IBMn Register Format

31	0
IBM	

Table 9-9 IBMn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
IBM	31:0	Instruction breakpoint address mask for condition: 0: Corresponding address bit not masked 1: Corresponding address bit masked	R/W	Undefined

9.2.8.4 Instruction Breakpoint ASID n (IBASIDn) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

The Instruction Breakpoint ASID n (IBASIDn) register has the ASID value used in the compare for instruction breakpoint n. The number of bits in the ASID field is 8, to match the ASID size in the TLB. This register is only valid for the 4Kc core.

IBASIDn Register Format

31	8	7	0
Res		ASID	

Table 9-10   IBASIDn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
Res	31:8	Must be written as zero; returns zero on read.	0	0
ASID	7:0	Instruction breakpoint ASID value for compare:	R/W	Undefined

9.2.8.5 Instruction Breakpoint Control n (IBCn) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

The Instruction Breakpoint Control n (IBCn) register controls setup of instruction breakpoint n.

IBCn Register Format

31 24	23 22		3	2	1	0
Res	ASID	Res		TE	Res	BE

Table 9-11   IBCn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	**Bits**			
Res	31:24	Must be written as zero; returns zero on read.	0	0
ASID	23	Use ASID value in compare for instruction breakpoint n: 0: Don't use ASID value in compare 1: Use ASID value in compare	4Kc core - R/W 4Km/4Kp cores - 0	Undefined
Res	22:3	Must be written as zero; returns zero on read.	0	0
TE	2	Use instruction breakpoint n as triggerpoint: 0: Don't use it as triggerpoint 1: Use it as triggerpoint	R/W	0
Res	1	Must be written as zero; returns zero on read.	0	0
BE	0	Use instruction breakpoint n as breakpoint: 0: Don't use it as breakpoint 1: Use it as breakpoint	R/W	0

9.2.9 Data Breakpoint Registers

The registers for data breakpoints are described below. These registers have implementation information and are used for setup the data breakpoints. All registers are in drseg, and the addresses are shown in section Table 9-12.

Table 9-12   Addresses for Data Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x2000	DBS	Data Breakpoint Status
0x2100 + 0x100 * n	DBAn	Data Breakpoint Address n
0x2108 + 0x100 * n	DBMn	Data Breakpoint Address Mask n
0x2110 + 0x100 * n	DBASIDn	Data Breakpoint ASID n
0x2118 + 0x100 * n	DBCn	Data Breakpoint Control n
0x2120 + 0x100 * n	DBVn	Data Breakpoint Value n
n is breakpoint number as 0 or 1		

An example of some of the registers; DBM0 is at offset 0x2108 and DBV1 is at offset 0x2220.

9.2.9.1 Data Breakpoint Status (DBS) Register

Compliance Level: Implemented only if any data breakpoints.

The Data Breakpoint Status (DBS) register holds implementation and status information about the instruction breakpoints.

The ASID applies to all the data breakpoints.

DBS Register Format

31	30	29 28	27	24 23		2	1	0
Res	ASID	Res	BCN		Res			BS

Table 9-13   DBS Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
Res	31	Must be written as zero; returns zero on read.	0	0
ASID	30	Indicates that ASID compare is supported in instruction breakpoints.	R	4Kc core - 1 4Km/4Kp cores - 0
Res	29:28	Must be written as zero; returns zero on read.	0	0
BCN	27:24	Number of instruction breakpoints implemented	R	2
Res	23:2	Must be written as zero; returns zero on read.	0	0
BS	1:0	Break status for breakpoint n is at BS[n], with n as 0 to 3. The bit is set to 1 when the condition for the corresponding breakpoint has matched.	R/W0	Undefined

9.2.9.2 Data Breakpoint Address n (DBAn) Register

Compliance Level: Implemented only for implemented data breakpoints.

The Data Breakpoint Address n (DBAn) register has the address used in the condition for data breakpoint n.

DBAn Register Format

31	0
DBA	

Table 9-14   DBAn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
DBA	31:0	Data breakpoint address for condition	R/W	Undefined

9.2.9.3 Data Breakpoint Address Mask n (DBMn) Register

Compliance Level: Implemented only for implemented data breakpoints.

The Data Breakpoint Address Mask n (DBMn) register has the mask for address compare used in the condition for data breakpoint n.

DBMn Register Format

31	0
DBM	

Table 9-15   DBMn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
DBM	31:0	Data breakpoint address mask for condition: 0: Corresponding address bit not masked 1: Corresponding address bit masked	R/W	Undefined

9.2.9.4 Data Breakpoint ASID n (DBASIDn) Register

Compliance Level: Implemented only for implemented data breakpoints.

The Data Breakpoint ASID n (DBASIDn) register has the ASID value used in the compare for data breakpoint n.

This register is only valid in the 4Kc core.

DBASIDn Register Format

31	8	7	0
Res		ASID	

Table 9-16   DBASIDn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Res	31:8	Must be written as zero; returns zero on read.	0	0
ASID	7:0	Data breakpoint ASID value for compare:	R/W	Undefined

9.2.9.5 Data Breakpoint Control n (DBCn) Register

Compliance Level: Implemented only for implemented data breakpoints.

The Data Breakpoint Control n (DBCn) register controls setup of data breakpoint n.

DBCn Register Format

31	24	23	22	18	17	14	13	12	11	8	7	4	3	2	1	0
Re		ASID	Res		BAI		NoSB	NoLB	Res		BLM		Res	TE	Res	BE

Table 9-17   DBCn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	**Bits**			
Res	31:24	Must be written as zero; returns zero on read.	0	0
ASID	23	Use ASID value in compare for data breakpoint n: 0: Don't use ASID value in compare 1: Use ASID value in compare	4Kc core - R/W 4Km/4Kp cores - 0	Undefined
Res	22:18	Must be written as zero; returns zero on read.	0	0
BAI	17:14	Byte access ignore controls ignore of access to specific byte. BAI[0] ignores access to byte at bits [7:0] of the data bus, BAI[1] ignores access to byte at bits [15:8], etc.: 0: Condition depends on access to corresponding byte 1: Access for corresponding byte is ignored	R/W	Undefined
NoSB	13	Controls if condition for data breakpoint is never fulfilled on a store transaction: 0: Condition may be fulfilled on store transaction 1: Condition is never fulfilled on store transaction	R/W	Undefined
NoLB	12	Controls if condition for data breakpoint is never fulfilled on a load transaction: 0: Condition may be fulfilled on load transaction 1: Condition is never fulfilled on load transaction	R/W	Undefined

Table 9-17 DBCn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
Res	11:8	Must be written as zero; returns zero on read.	0	0
BLM	7:4	Byte lane mask for value compare on data breakpoint. BLM[0] masks byte at bits [7:0] of the data bus, BLM[1] masks byte at bits [15:8], etc.: 0: Compare corresponding byte lane 1: Mask corresponding byte lane	R/W	Undefined
Res	3	Must be written as zero; returns zero on read.	0	0
TE	2	Use data breakpoint n as triggerpoint: 0: Don't use it as triggerpoint 1: Use it as triggerpoint	R/W	0
Res	1	Must be written as zero; returns zero on read.	0	0
BE	0	Use data breakpoint n as breakpoint: 0: Don't use it as breakpoint 1: Use it as breakpoint	R/W	0

9.2.9.6 Data Breakpoint Value n (DBVn) Register

Compliance Level: Implemented only for implemented data breakpoints.

The Data Breakpoint Value n (DBVn) register has the value used in the condition for data breakpoint n.

DBVn Register Format

31	0
DBV	

Table 9-18   DBVn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
DBV	31:0	Data breakpoint value for condition	R/W	Undefined

9.2.10 Test Access Port (TAP)

The following main features are supported by the TAP module:

- 5-pin industry standard JTAG Test Access Port (TCK, TMS, TDI, TDO, TRST_N) interface which are compatible with IEEE Std. 1149.1.

- Target chip and EJTAG feature identification available through the Test Access Port (TAP) controller.

- The processor can access external memory on the EJTAG Probe serially through the EJTAG pins. This is achieved through so-called Processor Access (PA), and is used to eliminate the use of the user's system memory for debug routines.

- Support for both ROM based debugger and debugging both through TAP.

9.2.11 EJTAG Internal and External Interfaces

The external interface of the EJTAG Module consists of the 5 signals defined by the IEEE standard.

Table 9-19   EJTAG Interface Pins

Pin	Type	Description
TCK	I	Test Clock Input Input clock used to shift data into or out of the Instruction or data registers. The TCK clock is independent of the processor clock, so the EJTAG probe can drive TCK independently of the processor clock frequency. The core signal for this is called EJ_TCK
TMS	I	Test Mode Select Input The TMS input signal is decoded by the TAP controller to control test operation. TMS is sampled on the rising edge of TCK. The core signal for this is called EJ_TMS
TDI	I	Test Data Input Serial input data (TDI) is shifted into the Instruction register or data registers on the rising edge of the TCK clock, depending on the TAP controller state. The core signal for this is called EJ_TDI

Table 9-19   EJTAG Interface Pins (continued)

Pin	Type	Description
TDO	O	Test Data Output Serial output data is shifted from the Instruction or data register to the TDO pin at the falling edge of the TCK clock. When no data is shifted out, the TDO is tri-stated. The core signal for this is called EJ_TDO with output enable control by EJ_TDOzstate.
TRST_N	I	Test Reset Input (Optional pin) The TRST_N pin is an active-low signal for asynchronous reset of the TAP controller and instruction in the TAP module, independent of the processor logic. The processor is not reset by the assertion of TRST_N. The core signal for this is called EJ_TRST_N This signal is optional, but power-on reset must apply a low pulse on this is signal at power-on and then leave it high, in case the signal is not available as a pin on the chip. If available on the chip, then it must be low on the board when the EJTAG debug features are unused by the probe.

9.3 Test Access Port Operation

The TAP controller is controlled by the Test Clock (TCK) and Test Mode Select (TMS) inputs. These two inputs determine whether an the Instruction register scan or data register scan is performed. The TAP consists of a small controller, driven by the TCK input, which responds to the TMS input as shown in the state diagram in Figure 9-3. The TAP uses both clock edges of TCK. TMS and TDI are sampled on the rising edge of TCK, while TDO changes on the falling edge of TCK.

At power-up the TAP is forced into the *Test-Logic-Reset* either by low value on TRST_N. The TAP instruction register is thereby reset to IDCODE. No other parts of the EJTAG hardware are reset through the *Test-Logic-Reset* state.

When test access is required, a protocol is applied via the TMS and TCK inputs, causing the TAP to exit the *Test-Logic-Reset* state and move through the appropriate states. From the *Run-Test/Idle* state, an Instruction register scan or a data register scan can be issued to transition the TAP through the appropriate states shown in Figure 9-3.

The states of the data and instruction register scan blocks are mirror images of each other adding symmetry to the protocol sequences. The first action that occurs when either block is entered is a capture operation. For the data

registers, the *Capture-DR* state is used to capture (or parallel load) the data into the selected serial data path. In the Instruction register, the *Capture-IR* state is used to capture status information into the Instruction register.

From the Capture states, the TAP transitions to either the Shift or Exit1 states. Normally the Shift state follows the Capture state so that test data or status information can be shifted out for inspection and new data shifted in. Following the Shift state, the TAP either returns to the *Run-Test/Idle* state via the Exit1 and Update states or enters the Pause state via Exit1. The reason for entering the Pause state is to temporarily suspend the shifting of data through either the Data or Instruction Register while a required operation, such as refilling a host memory buffer, is performed. From the Pause state shifting can resume by re-entering the Shift state via the Exit2 state or terminated by entering the *Run-Test/Idle* state via the Exit2 and Update states.

Upon entering the data or Instruction register scan blocks, shadow latches in the selected scan path are forced to hold their present state during the Capture and Shift operations. The data being shifted into the selected scan path is not output through the shadow latch until the TAP enters the Update-DR or Update-IR state. The Update state causes the shadow latches to update (or parallel load) with the new data that has been shifted into the selected scan path.
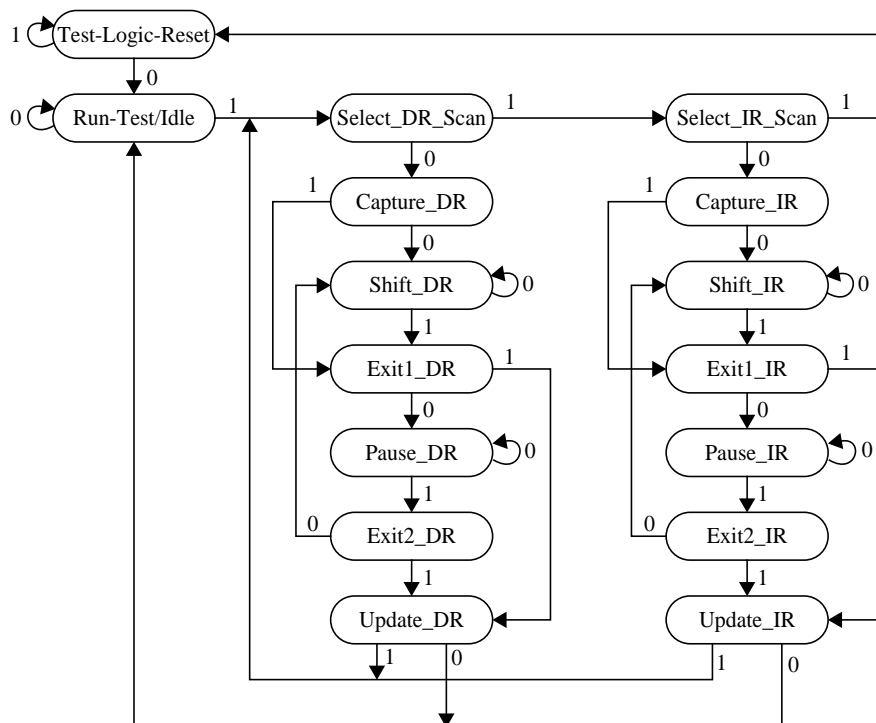
Figure 9-3   TAP Controller State diagram

9.3.1   Test-Logic-Reset State

In the *Test-Logic-Reset* state the boundary scan test logic is disabled. The test logic enters the *Test-Logic-Reset* state when the TMS input is held HIGH for at least five rising edges of TCK. The BYPASS instruction is forced into the instruction register output latches during this state. The controller remains in the *Test-Logic-Reset* state as long as TMS is HIGH.

9.3.2   Run-Test/Idle State

The controller enters the *Run-Test/Idle* state between scan operations. The controller remains in this state as long as TMS is held LOW. The instruction register and all test data registers retain their previous state. The instruction cannot change when the TAP controller is in this state.

When TMS is sampled HIGH at the rising edge of TCK, the controller transitions to the *Select_DR* state.

9.3.3   Select_DR_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If TMS is sampled LOW at the rising edge of TCK, the controller transitions to the *Capture_DR* state. A HIGH on TMS causes the controller to transition to the *Select_IR* state. The instruction cannot change while the TAP controller is in this state.

9.3.4   Select_IR_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If TMS is sampled LOW at the rising edge of TCK, the controller transitions to the *Capture_IR* state. A HIGH on TMS causes the controller to transition to the *Test-Reset-Logic* state. The instruction cannot change while the TAP controller is in this state.

9.3.5   Capture_DR State

In this state the boundary scan register captures value of the register addressed by the Instruction register, and the value is then shifted out in the *Shift_DR*. If TMS is sampled LOW at the rising edge of TCK, the controller transitions to the *Shift_DR* state. A HIGH on TMS causes the controller to transition to the *Exit1_DR* state. The instruction cannot change while the TAP controller is in this state.

9.3.6 Shift_DR State

In this state the test data register connected between TDI and TDO as a result of the current instruction shifts data one stage toward its serial output on the rising edge of TCK. If TMS is sampled LOW at the rising edge of TCK, the controller remains in the *Shift_DR* state. A HIGH on TMS causes the controller to transition to the *Exit1_DR* state. The instruction cannot change while the TAP controller is in this state.

9.3.7 Exit1_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If TMS is sampled LOW at the rising edge of TCK, the controller transitions to the *Pause_DR* state. A HIGH on TMS causes the controller to transition to the *Update_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

9.3.8 Pause_DR State

The Pause_DR state allows the controller to temporarily halt the shifting of data through the test data register in the serial path between TDI and TDO. All test data registers selected by the current instruction retain their previous state. If TMS is sampled LOW at the rising edge of TCK, the controller remains in the *Pause_DR* state. A HIGH on TMS causes the controller to transition to the Exit2_*DR* state. The instruction cannot change while the TAP controller is in this state.

9.3.9 Exit2_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If TMS is sampled LOW at the rising edge of TCK, the controller transitions to the *Shift_DR* state to allow another serial shift of data. A HIGH on TMS causes the controller to transition to the *Update_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

9.3.10 Update_DR State

When the TAP controller is in this state the value shifted in during the *Shift_DR* state takes effect at the rising edge of the TCK for the register indicated by the Instruction register.

If TMS is sampled LOW at the rising edge of TCK, the controller transitions to the *Run-Test/Idle* state. A HIGH on TMS causes the controller to transition to the *Select_DR_Scan* state. The instruction cannot change while the

TAP controller is in this state and all shift register stages in the test data registers selected by the current instruction retain their previous state.

9.3.11 Capture_IR State

In this state the shift register contained in the Instruction register loads a fixed pattern (00001_2) on the rising edge of TCK. The data registers selected by the current instruction retain their previous state.

If TMS is sampled LOW at the rising edge of TCK, the controller transitions to the *Shift_IR* state. A HIGH on TMS causes the controller to transition to the *Exit1_IR* state. The instruction cannot change while the TAP controller is in this state.

9.3.12 Shift_IR State

In this state the instruction register is connected between TDI and TDO and shifts data one stage toward its serial output on the rising edge of TCK. If TMS is sampled LOW at the rising edge of TCK, the controller remains in the *Shift_IR* state. A HIGH on TMS causes the controller to transition to the *Exit1_IR* state.

9.3.13 Exit1_IR State

This is a temporary controller state in which all registers retain their previous state. If TMS is sampled LOW at the rising edge of TCK, the controller transitions to the *Pause_IR* state. A HIGH on TMS causes the controller to transition to the *Update_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state and the instruction register retains its previous state.

9.3.14 Pause_IR State

The *Pause_IR* state allows the controller to temporarily halt the shifting of data through the instruction register in the serial path between TDI and TDO. If TMS is sampled LOW at the rising edge of TCK, the controller remains in the *Pause_IR* state. A HIGH on TMS causes the controller to transition to the Exit2_*IR* state. The instruction cannot change while the TAP controller is in this state.

9.3.15 Exit2_IR State

This is a temporary controller state in which the instruction register retains its previous state. If TMS is sampled LOW at the rising edge of TCK, the controller transitions to the *Shift_IR* state to allow another serial shift of data. A HIGH on TMS causes the controller to transition to the *Update_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

9.3.16 Update_IR State

The instruction shifted into the instruction register takes effect on the rising edge of TCK.

If TMS is sampled LOW at the rising edge of TCK, the controller transitions to the *Run-Test/Idle* state. A HIGH on TMS causes the controller to transition to the *Select_DR_Scan* state.

9.4 Test Access Port (TAP) Instructions

The TAP Instruction register allows instructions to be serially input into the device when TAP controller is in the Shift-IR state. Instructions are decoded and define the serial test data register path that is used to shift data between TDI and TDO during data register scanning.

The Instruction register is a 5-bit register. In the current EJTAG implementation only some instructions have been decoded; the unused instructions are set default to the BYPASS instruction.

Table 9-20   Implemented EJTAG instructions

Value	Instruction	Function
0x01	IDCODE	Select Chip Identification data register
0x03	IMPCODE	Select Implementation Register
0x08	ADDRESS	Select Address register
0x09	DATA	Select Data register
0x0A	CONTROL	Select EJTAG Control register
0x0B	ALL	Select the Address, Data and EJTAG Control registers
0x0C	EJTAGBOOT	Set EjtagBrk, ProbEn and ProbTrap to 1 as reset value

Table 9-20   Implemented EJTAG instructions

Value	Instruction	Function
0x0D	NORMALBOOT	Set EjtagBrk, ProbEn and ProbTrap to 0 as reset value
0x1F	BYPASS	Bypass mode

9.4.1   BYPASS Instruction

The required BYPASS instruction allows the processor to remain in a functional mode and selects the Bypass register to be connected between TDI and TDO. The BYPASS instruction allows serial data to be transferred through the processor from TDI to TDO without affecting its operation. The bit code of this instruction is defined to be all ones by the IEEE 1149.1 standard. Any unused instruction is defaulted to the BYPASS instruction.

9.4.2   IDCODE Instruction

The IDCODE instruction allows the processor in its functional mode and selects the Device Identification (ID) register to be connected between TDI and TDO. The Device ID register is a 32- bit shift register containing information regarding the IC manufacturer, device type, and version code. Accessing the Identification Register does not interfere with the operation of the processor. Also, access to the Identification Register is immediately available, via a TAP data scan operation, after power-up when the TAP has been reset with on-chip power-on or through the optional TRST_N pin.

9.4.3   IMPCODE Instruction

This instruction selects the Implementation register for output, which is always 32 bit.

9.4.4   ADDRESS Instruction

This instruction is used to select the Address register to be connected between TDI and TDO. The EJTAG Probe shifts 32-bits through the TDI pin into the Address register and shifts out the captured address via the TDO pin.

9.4.5   DATA Instruction

This instruction is used to select the Data register to be connected between TDI and TDO. The EJTAG Probe shifts 32-bits of TDI data into the Data register and shifts out the captured data via the TDO pin.

9.4.6   CONTROL Instruction

This instruction is used to select the EJTAG Control register to be connected between TDI and TDO. The EJTAG Probe shifts 32- bits of TDI data into the EJTAG Control register and shifts out the EJTAG Control register bits via TDO.

9.4.7   ALL Instruction

This instruction is used to select the concatenation of the Address and Data register, and the EJTAG Control register between TDI and TDO. It can be used in particular if switching instructions in the instruction register takes too many TCK cycles. The first bit shifted out is bit 0.
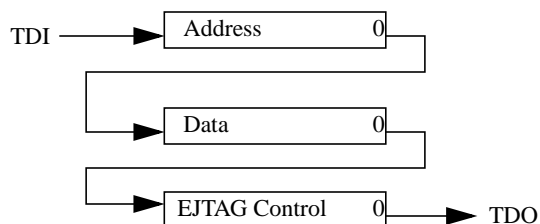
Figure 9-4   Concatenation of the EJTAG Address, Data and Control Registers

9.4.8   EJTAGBOOT Instruction

When the EJTAGBOOT instruction is given and Update-IR state is left, then the reset value of the ProbTrap, ProbEn and EjtagBrk bits in the EJTAG Control register are set to 1 after hard or soft reset.

This EJTAGBOOT indication is effective until NORMALBOOT instruction is given, TRST_N is asserted or rising edge of TCK occurs when TAP controller is in Test-Logic-Reset state.

It is thereby possible to make the CPU go into debug mode just after hard or soft reset, without fetching or executing any instructions from the normal memory area. This can be used for download of code to a system which have no code in ROM.

The Bypass register is selected when the EJTAGBOOT instruction is given.

9.4.9   NORMALBOOT Instruction

When the NORMALBOOT instruction is given and Update-IR state is left, then the reset value of the ProbTrap, ProbEn and EjtagBrk bits in the EJTAG Control register are set to 0 after hard or soft reset.

The Bypass register is selected when the NORMALBOOT instruction is given.

9.5 EJTAG Registers

The EJTAG TAP Module has the following registers accessible through the TAP:

- Instruction Register
- Data Registers Overview
- Bypass Register
- Device Identification Register
- Implementation Register
- EJTAG Control Register (ECR)
- Processor Access Address Register
- Processor Access Data Register

9.5.1   Instruction Register

The Instruction register is accessed when the TAP receives an Instruction register scan protocol. During an Instruction register scan operation the TAP controller selects the output of the Instruction register to drive the TDO pin. The shift register consists of a series of bits arranged to form a single scan path between TDI and TDO. During an Instruction register scan operations, the TAP controls the register to capture status information and shift data from TDI to TDO. Both the capture and shift operations occur on the rising edge of TCK. However, the data shifted out from the TDO occurs on the falling edge of TCK. In the Test-Logic-Reset and Capture-IR state, the

instruction shift register is set to 00001_2, as for IDCODE instruction. This forces the device into the functional mode and selects the Device ID register. The Instruction register is 5 bits wide. The instruction shifted in takes effect for the following data register scan operation.

9.5.2 Data Registers Overview

The EJTAG uses several data registers, which are arranged in parallel from the primary TDI input to the primary TDO output. The Instruction register supplies the address that allows one of the data registers to be accessed during a data register scan operation. During a data register scan operation, the addressed scan register receives TAP control signals to capture the register and shift data from TDI to TDO. During a data register scan operation, the TAP selects the output of the data register to drive the TDO pin. The register is updated in the Update-DR state with respect to write bits.

This description applies in general to the following data registers.

9.5.3 Bypass Register

The *Bypass* register consists of a single scan register bit. When selected, the Bypass register provides a single bit scan path between TDI and TDO. The Bypass register allows abbreviating the scan path through devices that are not involved in the test. The Bypass register is selected when the Instruction register is loaded with a pattern of all ones to satisfy the IEEE 1149.1 Bypass instruction requirement.

9.5.4 Device Identification (ID) Register

The *Device Identification* register is defined by IEEE 1149.1, to identify the device's manufacturer, part number, revision, and other device-specific information. Table 9-21 shows the bit assignments defined for the read-only Device Identification Register, and inputs to the core determine the value of these bits. These bits can be scanned out of the ID register after being selected. The register is selected when the Instruction register is loaded with the IDCODE instruction.

Table 9-21   Device Identification Register

Bit(s)	Mnemonic	Description	R/W	Reset
31:28	Version	**Version** (4 bits) This field identifies the version number of the processor derivative.	R	EJ_Version[3:0]

Table 9-21   Device Identification Register

Bit(s)	Mnemonic	Description	R/W	Reset
27:12	PartNumber	**Part Number** (16 bits) This field identifies the part number of the processor derivative.	R	EJ_PartNumber[15:0]
11:1	ManufID	**Manufacturer Identity** (11 bits) Accordingly to IEEE 1149.1-1990, the manufacturer identity code shall be a compressed form of the JEDEC Publications 106-A.	R	EJ_ManufID[10:0]
0	reserved	reserved	R	1

9.5.5 Implementation Register

This 32-bit read-only register is used to identify the features of the EJTAG implementation. Some of the reset value are set by inputs to the core.The register is selected when the Instruction register is loaded with the IMPCODE instruction.

Table 9-22   Implementation Register Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
EJTAGver	31:29	EJTAG Version	R	1
reserved	28:25	reserved	R	0
DINTsup	24	DINT Signal Supported from Probe This bit indicates if the DINT signal from the probe is supported: 0: DINT signal from the probe is not supported 1: Probe can use DINT signal to make debug interrupt.	R	EJ_DINTsup
ASIDsize	23:21	Size of ASID field in implementation This is determined by the EJ_ASIDused signal to the core. No ASID in implementation: EJ_ASIDused should be set to 0. 8-bit ASID in implementation: EJ_ASIDused should be set to 1.	R	See description
reserved	20:15	reserved	R	0
NoDMA	14	No EJTAG DMA Support	R	1
reserved	13:0	reserved	R	0

9.5.6 EJTAG Control Register

This 32-bit register controls the various operations of the TAP modules. This register is selected by shifting in the CONTROL instruction. Bits in the EJTAG Control register can be set/cleared by shifting in data; status is read by shifting out the contents of this register. This EJTAG Control register can only be accessed by the TAP interface.

The EJTAG Control register is not updated in the Update-DR state unless the Reset occurred (Rocc), bit 31, is either 0 or written to 0. This is in order to ensure prober handling of processor accesses.

The value used for reset indicated in the table below takes effect on both hard and soft CPU reset, but no on TAP controller reset by e.g. TRST_N. TCK clock is not required when the hard or soft CPU reset occurs, but the bits are still updated to the reset value when the TCK applies. The first 5 TCK clocks after hard or soft CPU reset may result in reset of the bits, due to synchronization between clock domains.

Table 9-23   EJTAG Control Register Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
Rocc	31	Reset Occurred The bit indicates if hard or soft reset has occurred: 0: No reset occurred since bit last cleared. 1: Reset occurred since bit last cleared. The Rocc bit will keep the 1 value as long as hard or soft reset is applied. This bit must be cleared by the probe, to acknowledge that the incident was detected. The EJTAG Control register is not updated in the Update-DR state unless Rocc is 0, or written to 0. This is in order to ensure prober handling of processor access.	R/W	1

Table 9-23   EJTAG Control Register Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
Psz[1:0]	30:29	Processor Access Transfer Size These bits are used in combination with the lower two address bits of the Address register to determine the size of a processor access transaction. The bits are only valid when processor access is pending.	R	Undefined
Res	28:23	reserved	R	0
Doze	22	Doze state The Doze bit indicates any kind of low power mode. The value is sampled in the Capture-DR state of the TAP controller: 0: CPU not in low power mode. 1: CPU is in low power mode Doze includes the Reduced Power (RP) and WAIT power-reduction modes.	R	n.a.

Within the Psz description cell, the following sub-table and note appear:

PA[1:0]	Psz[1:0]	Transfer Size
00	00	Byte (LE, byte 0; BE, byte 3)
01	00	Byte (LE, byte 1; BE, byte 2)
10	00	Byte (LE, byte 2; BE, byte 1)
11	00	Byte (LE, byte 3; BE, byte 0)
00	01	Halfword (LE, bytes 1:0; BE, bytes 3:2)
10	01	Halfword (LE, bytes 3:2; BE, bytes 1:0)
00	10	Word (LE, BE; bytes 3, 2, 1, 0)
00	11	Triple (LE, bytes 2, 1, 0; BE, bytes 3, 2,1)
01	11	Triple (LE, bytes 3, 2, 1; BE, bytes 2, 1, 0)
All others		Reserved

Note: LE=little endian, BE=big endian, the byte# refers to the byte number in a 32-bit register, where byte 3 = bits 31:24; byte 2 = bits 23:16; byte 1 = bits 15:8; byte 0=bits 7:0, independently of the need.

Table 9-23   EJTAG Control Register Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
Halt	21	Halt state The Halt bit indicates if the internal system bus clock is running or stopped. The value is sampled in the Capture-DR state of the TAP controller: 0: Internal system clock is running 1: Internal system clock is stopped	R	n.a.
PerRst	20	Peripheral Reset When the bit is set to 1, it is only guaranteed that the peripheral reset has occurred in the system when the read value of this bit is also 1. This is to ensure that the setting from the TCK clock domain gets effect in the CPU clock domain, and in peripherals. When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is cleared in the CPU clock domain also. This bit controls the EJ_PerRst signal on the core.	R/W	0
PRnW	19	Processor Access Read and Write This bit indicates if the pending processor access is for a read or write transaction, and the bit is only valid while PrAcc is set: 0: Read transaction 1: Write transaction	R	Undef.

Table 9-23 EJTAG Control Register Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
PrAcc	18	Processor Access (PA) Read value of this bit indicates if a Processor Access (PA) to the EJTAG memory is pending: 0: No pending processor access 1: Pending processor access The probe's software must clear this bit to 0 to indicate the end of the PA. Write of 1 is ignored. A pending PA is cleared when Rocc is set, but another PA may occur just after the reset if a debug exception occurs. Finishing a PA is not accepted while the Rocc bit is set. This is to avoid that a PA occurring after the reset is finished due to indication of a PA that occurred before the reset.	R/W0	0
Res	17	reserved	R	0
PrRst	16	Processor Reset (Implementation dependent behavior) When the bit is set to 1, then it is only guaranteed that this setting has taken effect in the system when the read value of this bit is also 1. This is to ensure that the setting from the TCK clock domain gets effect in the CPU clock domain, and in peripherals. When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is cleared in the CPU clock domain also. This bit controls the EJ_PerRst signal. If the signal is used in the system, then it must be ensured that both the processor and all devices required for a reset are properly reset. Otherwise the system may fail or hang. The bit resets itself, since the EJTAG Control register is reset by hard or soft reset.	R/W	0

Table 9-23   EJTAG Control Register Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
ProbEn	15	Probe Enable This bit indicates to the CPU if the EJTAG memory is handled by the probe so processor accesses are answered: 0: The probe does not handle EJTAG memory transactions 1: The probe does handle EJTAG memory transactions It is an error by the software controlling the probe if it sets the ProbTrap to 1 but the ProbEn to 0. The operation of the processor is UNDEFINED in this case. The ProbEn bit is reflected as a read-only bit in the ProbEn bit, bit 0, in the Debug Control Register (DCR). The read value indicates the effective value in the DCR, due to synchronization issues between TCK and CPU clock domains. However, it is ensured that change of the ProbEn prior to setting the EjtagBrk bit will have effect for the debug handler executed due to the debug exception. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	R/W	0 or 1 from EJTAGBOOT

Table 9-23   EJTAG Control Register Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
ProbTrap	14	Probe Trap This bit controls the location of the debug exception vector: 0: In normal memory 0xBFC0.0480 1: In EJTAG memory at 0xFF20.0200 in dmseg Valid setting of the ProbTrap bit depends on the setting of the ProbEn bit, see comment under ProbEn bit. The ProbTrap should not be set to 1, for debug exception vector in EJTAG memory, unless the ProbEn bit is also set to 1 to indicate that the EJTAG memory may be accessed. The read value indicates the effective value to the CPU, due to synchronization issues between TCK and CPU clock domains. However, it is ensured that change of the ProbTrap prior to setting the EjtagBrk bit will have effect for the EjtagBrk. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	R/W	0 or 1 from EJTAGBOOT
Res	13	reserved	R	0
EjtagBrk	12	EJTAG Break Setting this bit to 1 causes a debug exception to the processor, unless the CPU was in debug mode or another debug exception occurred. When the debug exception occurs, the processor core clock is restarted if the CPU was in low power mode. This bit is cleared by hardware when the debug exception is taken. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	W1/R	0 or 1 from EJTAGBOOT
Res	11:4	reserved	R	0

Table 9-23   EJTAG Control Register Descriptions (continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
BrkSt	3	Break Status This bit indicates the debug or non-debug mode: 0: Processor is in non-debug mode 1: Processor is in debug mode The bit is sampled in the Capture-DR state of the TAP controller.	R	0
Res	2:0	reserved	R	0

9.5.7 Processor Access Address Register

The Processor Access Address (PAA) register is used to provide the address of the processor access in the dmseg, and the register is only valid when a processor access is pending. The length of the Address register is 32 bits, and this register is selected by shifting in the ADDRESS instruction.

9.5.8 Processor Access Data Registers

The Processor Access Data (PAD) register is used to provide data value to and from a processor access. The length of the Address register is 32 bits, and this register is selected by shifting in the DATA instruction.

The register has the written value for a processor access write due to a CPU store to the dmseg, and the output from this register is only valid when a processor access write is pending. The register is used to provide the data value for processor access read due to a CPU load or fetch from the dmseg, and the register should only be updated with a new value when a processor access write is pending.

The PA Data register is 32 bits wide. Data alignment is not used for this register, so the value in the PAD register matches data on the internal bus. The undefined bytes for a PA write are undefined, and for a PAD read then 0 (zero) must be shifted in for the unused bytes.

The organization of bytes in the PAD register depends on the endianess of the core, as shown in Figure 9-5. The endian mode for debug/kernel mode is determined by the state of the *EB_Endian* input at power-up.
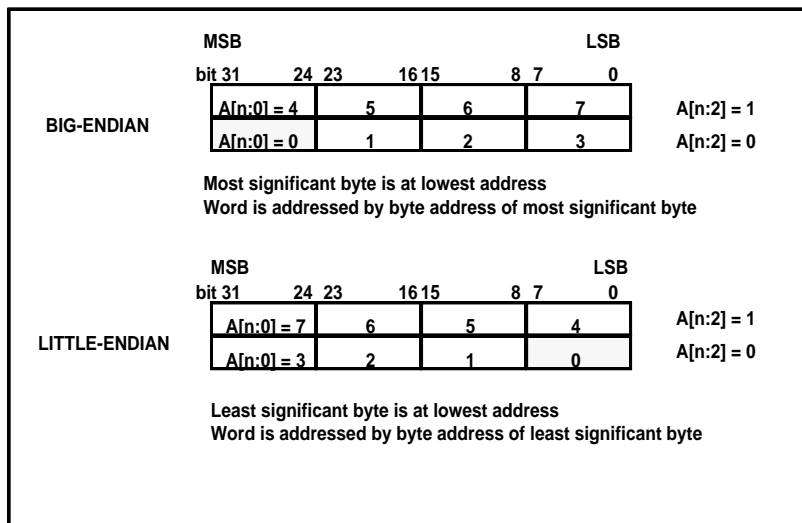
Figure 9-5   Endian Formats for the PA Data Registers

The size of the transaction and thus the number of bytes available/required for the PA Data register is determined by the Psz field in the ECR.

9.6 Processor Accesses

The TAP modules support handling of fetch, load and store from the CPU through the dmseg segment, whereby the TAP module can operate like a is a *slave unit* connected to the on-chip bus. The core can then execute code taken from the EJTAG Probe and it can access data (via load or store) which is located on the EJTAG Probe. This occurs in a serial way through the EJTAG interface: the core can thus execute instructions e.g. debug monitor code, without occupying the user's memory.

Accessing the dmseg segment (EJTAG memory) can only occur when the processor accesses an address in the range from 0xFF20.0000 to 0xFF2F.FFFF, the ProbEn bit is set, and the processor is in debug mode (DM=1). In addition the LSNM bit in the CP0 Debug register controls transactions to/from the dmseg.

When a debug exception is taken, while the ProbTrap bit is set, the processor will start fetching instructions from address 0xFF20.0200.

A pending processor access can only finish if the probe writes 0 to PrAcc or by soft or hard reset.

9.6.1 Fetch/Load and Store from/to the EJTAG Probe through dmseg

1. The internal hardware latches the requested address into the PA Address register (in case of the Debug exception: 0xFF20-0200).

2. The internal hardware sets the following bits in the EJTAG Control register:
   PrAcc = 1 (selects Processor Access operation)
   PRnW = 0 (selects processor read operation)
   Psz[1:0] = value depending on the transfer size

3. The EJTAG Probe selects the EJTAG Control register, shifts out this control register's data and tests the PrAcc status bit (Processor Access): when the PrAcc bit is found 1, it means that the requested address is available and can be shifted out.

4. The EJTAG Probe checks the PRnW bit to determine the required access.

5. The EJTAG Probe selects the PA Address register and shifts out the requested address.

6. The EJTAG Probe selects the PA Data register and shifts in the instruction corresponding to this address.

7. The EJTAG Probe selects the EJTAG Control register and shifts a PrAcc = 0 bit into this register to indicate to the processor that the instruction is available.

8. The instruction becomes available in the instruction register and the processor starts executing.

9. The processor increments the program counter and outputs an instruction read request for the next instruction. This starts the whole sequence again.

Using the same protocol, the processor can also execute a load instruction to access the EJTAG Probe's memory. For this to happen, the processor must execute a load instruction (e.g. a LW, LH, LB) with the target address in the appropriate range.

Almost the same protocol is used to execute a store instruction to the EJTAG Probe's memory through dmseg. The store address must be in the range: 0xFF20-0000 to 0xFF2F-FFFF, the ProbEn bit must be set and the processor has to be in debug mode (DM=1). The sequence of actions is found below:

1. The internal hardware latches the requested address into the PA Address register

2. The internal hardware latches the data to be written into the PA Data register.

3. The internal hardware sets the following bits in the EJTAG Control register:
   PrAcc = 1 (selects Processor Access operation)
   PRnW = 1 (selects processor write operation)
   Psz[1:0] = value depending on the transfer size

4. The EJTAG Probe selects the EJTAG Control register, shifts out this control register's data and tests the PrAcc status bit (Processor Access): when the PrAcc bit is found 1, it means that the requested address is available and can be shifted out.

5. The EJTAG Probe checks the PRnW bit to determine the required access.

6. The EJTAG Probe selects the PA Address register and shifts out the requested address.

7. The EJTAG Probe selects the PA Data register and shifts out the data to be written.

8. The EJTAG Probe selects the EJTAG Control register and shifts a PrAcc = 0 bit into this register to indicate to the processor that the write access is finished.

9. The EJTAG Probe writes the data to the requested address in its memory.

10. The processor detects that PrAcc bit = 0, which means that it is ready to handle a new access.

The above examples implies that no reset occurs during the operations, and that Rocc is cleared.

Instruction Set Overview

This chapter provides a general overview on the three CPU instruction set formats of the MIPS architecture: Immediate, Jump, and Register. Refer to Chapter 11 for a complete listing and description of instructions.

This chapter discusses the following topics

- Section 10.1, "CPU Instruction Formats"

- Section 10.2, "Load and Store Instructions"

- Section 10.3, "Computational Instructions"

- Section 10.4, "Jump and Branch Instructions"

- Section 10.5, "Control Instructions"

- Section 10.6, "Coprocessor Instructions"

- Section 10.7, "Enhancements to the MIPS Architecture"

10.1 CPU Instruction Formats

Each CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats—immediate (I-type), jump (J-type), and register (R-type)—as shown in Figure 10-1. The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed.

I-Type (Immediate)

31 26	25 21	20 16	15 0
op	rs	rt	immediate

J-Type (Jump)

31 26	25 0
op	target

R-Type (Register)

31 26	25 21	20 16	15 11	10 6	5 0
op	rs	rt	rd	sa	funct
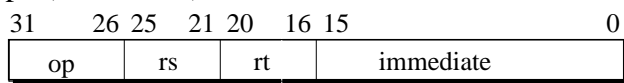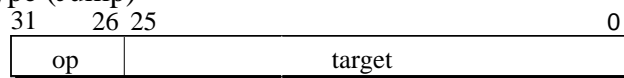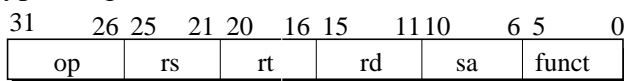
op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination) register or branch condition
immediate	16-bit immediate value, branch displacement or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
sa	5-bit shift amount
funct	6-bit function field

Figure 10-1 Instruction Formats

10.2 Load and Store Instructions

Load and store are immediate (I-type) instructions that move data between memory and the general registers. The only addressing mode that load and store instructions directly support is *base register plus 16-bit signed immediate offset*.

10.2.1 Scheduling a Load Delay Slot

A load instruction that does not allow its result to be used by the instruction immediately following is called a *delayed load instruction*. The instruction slot immediately following this delayed load instruction is referred to as the *load delay slot*.

In all the 4K cores, the instruction immediately following a load instruction can use the contents of the loaded register, however in such cases hardware interlocks insert additional real cycles. Although not required, the scheduling of load delay slots can be desirable, both for performance and R-Series processor compatibility.

10.2.2 Defining Access Types

Access type indicates the size of a core data item to be loaded or stored, set by the load or store instruction opcode.

Regardless of access type or byte ordering (endianness), the address given specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the three low-order bits of the address, define the bytes accessed within the addressed word as shown in Table 10-1. Only the combinations shown in Table 10-1 are permissible; other combinations cause address error exceptions.

Table 10-1 Byte Access within a Word

Access Type Mnemonic (*Value*)	Low Order Address Bits			Bytes Accessed							
				Big Endian (Byte)				Little Endian (Byte)			
	2	1	0	0	1	2	3	3	2	1	0
Word (*3*)	0	0	0	0	1	2	3	3	2	1	0
Triplebyte (*2*)	0	0	0	0	1	2			2	1	0
	0	0	1		1	2	3	3	2	1	
Halfword (*1*)	0	0	0	0	1					1	0
	0	1	0			2	3	3	2		
Byte (*0*)	0	0	0	0							0
	0	0	1		1					1	
	0	1	0			2			2		
	0	1	1				3	3			

10.3 Computational Instructions

Computational instructions can be either in register (R-type) format, in which both operands are registers, or in immediate (I-type) format, in which one operand is a 16-bit immediate.

Computational instructions perform the following operations on register values:

- – Arithmetic
- – Logical
- – Shift
- – Multiply

– Divide

These operations fit in the following four categories of computational instructions:

– ALU Immediate instructions

– Three-operand Register-type Instructions

– Shift Instructions

– Multiply And Divide Instructions

10.3.1 Cycle Timing for Multiply and Divide Instructions

Any multiply instruction in the integer pipeline is transferred to the multiplier as remaining instructions continue through the pipeline; the product of the multiply instruction is saved in the HI and LO registers. If the multiply instruction is followed by an MFHI or MFLO before the product is available, the pipeline interlocks until this product does become available. Refer to Chapter 2 for more information on instruction latency and repeat rates.

10.4 Jump and Branch Instructions

Jump and branch instructions change the control flow of a program. All jump and branch instructions occur with a delay of one instruction: that is, the instruction immediately following the jump or branch (this is known as the instruction in the *delay slot*) always executes while the target instruction is being fetched from storage.

10.4.1 Overview of Jump Instructions

Subroutine calls in high-level languages are usually implemented with Jump or Jump and Link instructions, both of which are J-type instructions. In J-type format, the 26-bit target address shifts left 2 bits and combines with the high-order 4 bits of the current program counter to form an absolute address.

Returns, dispatches, and large cross-page jumps are usually implemented with the Jump Register or Jump and Link Register instructions. Both are R-type instructions that take the 32-bit byte address contained in one of the general purpose registers.

For more information about jump instructions, refer to the individual instructions in Section 10.6.

10.4.2 Overview of Branch Instructions

All branch instruction target addresses are computed by adding the address of the instruction in the delay slot to the 16-bit *offset* (shifted left 2 bits and sign-extended to 32 bits). All branches occur with a delay of one instruction.

If a conditional branch likely is not taken, the instruction in the delay slot is nullified.

Branches, jumps, ERET, and DERET instructions should not be placed in the delay slot of a branch or jump.

10.5 Control Instructions

Control instructions allow the software to initiate traps; they are always R-type.

10.6 Coprocessor Instructions

CP0 instructions perform operations on the System Control Coprocessor registers to manipulate the memory management and exception handling facilities of the processor. Refer to Chapter 11 for a listing of CP0 instructions.

10.7 Enhancements to the MIPS Architecture

The core execution unit implements the MIPS32™ architecture, which includes the following instructions.

- CLOCount Leading Ones
- CLZCount Leading Zeros
- MADDMultiply and Add Word
- MADDUMultiply and Add Unsigned Word
- MSUBMultiply and Subtract Word
- MSUBUMultiply and Subtract Unsigned Word
- MULMultiply Word to Register
- SSNOPSuperscalar Inhibit NOP

10.7.1 CLO - Count Leading Ones

The CLO instruction counts the number of leading ones in a word. The 32-bit word in the GPR *rs* is scanned from most-significant to least-significant bit. The number of leading ones is counted and the result is written to the GPR *rd*. If all 32 bits are set in the GPR *rs*, the result written to the GPR *rd* is 32.

10.7.2 CLZ - Count Leading Zeros

The CLZ instruction counts the number of leading zeros in a word. The 32-bit word in the GPR *rs* is scanned from most-significant to least-significant bit. The number of leading zeros is counted and the result is written to the GPR *rd*. If all 32 bits are cleared in the GPR *rs*, the result written to the GPR *rd* is 32.

10.7.3 MADD - Multiply and Add Word

The MADD instruction multiplies two words and adds the result to the HI/LO register pair. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is added to the 64-bit concatenated values in the HI and LO register pair. The resulting value is then written back to the HI and LO registers. No arithmetic exception occurs under any circumstances.

10.7.4 MADDU - Multiply and Add Unsigned Word

The MADDU instruction multiplies two unsigned words and adds the result to the HI/LO register pair. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is added to the 64-bit concatenated values in the HI and LO register pair. The resulting value is then written back to the HI and LO registers. No arithmetic exception occurs under any conditions.

10.7.5 MSUB - Multiply and Subtract Word

The MSUB instruction multiplies two words and subtracts the result from the HI/LO register pair. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values in the HI and LO register pair. The resulting value is then written back to the HI and LO registers. No arithmetic exception occurs under any circumstances.

10.7.6 MSUBU - Multiply and Subtract Unsigned Word

The MSUBU instruction multiplies two unsigned words and subtracts the result from the HI/LO register pair. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values in the HI and LO register pair. The resulting value is then written back to the HI and LO registers. No arithmetic exception occurs under any circumstances.

10.7.7 MUL - Multiply Word

The MUL instruction multiplies two words and writes the result to a GPR. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The least-significant 32-bits of the product are written to the GPR *rd*. The contents of the HI and LO register pair are not defined after the operation. No arithmetic exception occurs under any circumstances.

10.7.8 SSNOP- Superscalar Inhibit NOP

The MIPS32 4K™ processor cores treat this instruction as a regular NOP.

MIPS32 4K™ Processor Core Instructions

This chapter provides a detailed guide to understanding the instruction set for the MIPS32 4K™ processor cores, which is a subset of the MIPS32 architecture. The chapter is divided into the following sections:

- Section 11.1, "Understanding the Instruction Fields"
- Section 11.2, "Instruction Hazards"
- Section 11.3, "CPU Opcode Map"
- Section 11.4, "Instruction Set"

11.1 Understanding the Instruction Fields

Figure 11-1 shows an example instruction. Following the figure are descriptions of the fields listed below. Some or all of these field appear in the description of each instruction.

- "Instruction Fields"
- "Instruction Descriptive Name and Mnemonic"
- "Format Field"
- "Purpose Field"
- "Description Field"
- "Restrictions Field"
- "Operation Field"
- "Exceptions Field"

Instruction mnemonic
and descriptive name ➤

Example Instruction Name   **EXAMPLE**

Instruction encoding
constant and variable
field names and values

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

Architecture level at
which instruction was
defined/redefined and
assembler format(s)
for each definition

Format: EXAMPLE rd, rs, rt   **MIPS I**

Short description ➤ **Purpose:** to execute an EXAMPLE op

Symbolic description ➤ **Description:** rd ← rs exampleop rt

Full description of
instruction operation

This section describes the operation of the instruction in text, tables, and illustrations. It
includes information that would be difficult to encode in the Operation section.

Restrictions on
instruction and
operands

Restrictions: This section lists any restrictions for the instruction. This can include values
of the instruction encoding fields such as register specifiers, operand values, operand
formats, address alignment, instruction scheduling hazards, and type of memory access for
addressed locations.

High-level language
description of
instruction operation

Operation
```
/* This section describes the operation of an instruction in a */
/* high-level pseudo-language. It is precise in ways that the */
/* Description section is not, but is also missing information */
/* that is hard to express in pseudocode.*/
temp← GPR[rs] exampleop GPR[rt]
GPR[rd]← sign_extend(temp31..0)
```

Exceptions that
instruction can cause

Exceptions: A list of exceptions taken by the instruction

Notes for programmers **Programming Notes:** Information useful to programmers, but not necessary to describe
the operation of the instruction

Figure 11-1 Example Instruction Description

11.1.1 Instruction Fields

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

- The values of constant fields and the *opcode* names for *opcode* fields are listed in uppercase (SPECIAL and ADD in Figure 11-2).

- All variable fields are listed with the lowercase names used in the instruction description (*rs*, *rt* and *rd* in Figure 11-2).

- Fields that contain zeros but are not named are unused fields that are required to be zero (bits 10:6 in Figure 11-2).

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL	rs	rt	rd	0	ADD
0 0 0 0 0 0				0 0 0 0 0	1 0 0 0 0 0
6	5	5	5	5	6
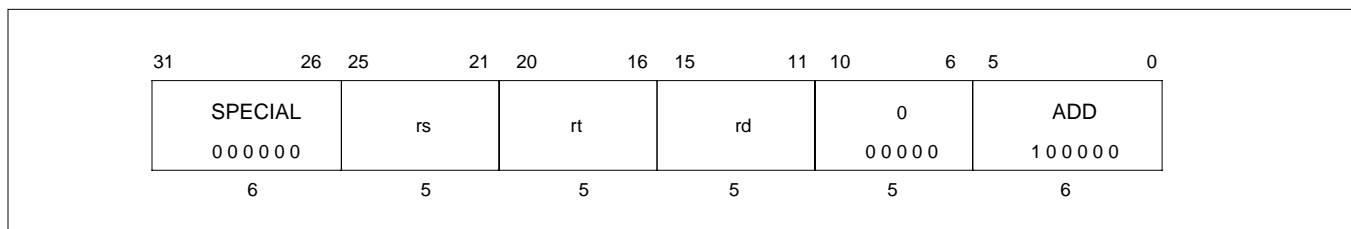
Figure 11-2   Example of Instruction Fields

11.1.2 Instruction Descriptive Name and Mnemonic

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

Add Word   **ADD**

11.1.3 Format Field

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

Format: ADD rd, rs, rt	**MIPS I**

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

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

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

11.1.4 Purpose Field

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

Purpose: to add 32-bit integers. If overflow occurs, then trap.

11.1.5 Description Field

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

Description: rd ← rs + rt

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

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

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

11.1.6 Restrictions Field

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

- Valid values for instruction fields (for example, see BGEZAL)

- Alignment requirements for memory addresses (for example, see LW)

- Valid values of operands (for example, see DIV)

- ~~Valid operand formats (for example, see floating point ADD.fmt)~~ (Floating Point only)

- Order of instructions necessary to guarantee correct execution. These ordering constraints avoid pipeline hazards for which some processors do not have hardware interlocks (for example, see ERET).

- Valid memory access types (for example, see LL/SC)

11.1.7 Operation Field

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

```
Operation:
     temp      ←    GPR[rt]31..0
     FCC[0]    ←    GPR[rt]31..0
```

11.1.8 Exceptions Field

The *Exceptions* field lists the exceptions that can be caused by *Operation* of the instruction. It omits exceptions that can be caused by the instruction fetch, for instance, TLB Refill, and also omits exceptions that can be caused by asynchronous external events such as an Interrupt.

```
Exceptions:
     Integer Overflow
```

11.2 Instruction Hazards

In general, the core ensures that instructions are executed following a fully sequential program model. Each instruction in the program sees the results of the previous instruction. There are some exceptions to this model. These exceptions are referred to as *instruction hazards*.

The following table shows the instruction hazards that exist in the core. The first and second instruction fields indicate the combination of instructions that do not ensure a sequential programming model. The Spacing field indicates the number of unrelated instructions (such as NOPs or SSNOPs) that should be placed between the first and second instructions of the hazard in order to ensure that the effects of the first instruction are seen by the second instruction. Entries in the table that are listed as 0 are traditional MIPS hazards which are not hazards on the 4K cores. (MT Compare to Timer Interrupt cleared is system dependent since Timer Interrupt is an output of

the core that can be returned to the core on one of the SI_Int pins. This number is the minimum time due to going through the core's I/O registers. Typical implementations will not add any latency to this).

Table 11-1   Instruction Hazards

Instruction Hazards		
First Instruction	**Second Instruction**	**Spacing (Instructions)**
Watch Register Write	Instruction Fetch Matching Watch Register	2
	Load/Store Reference Matching Watch Register	0
TLBWI/TLBWR	Instruction fetch affected by new page mapping	3
	Load/Store affected by new page mapping	0
	TLBP/TLBR	0
TLBR	Move from Coprocessor Zero Register	0
Move to EntryHI	TLBWR/TLBWI/TLBP	1
Move to EntryLo0 or EntryLo1	TLBWR/TLBWI	0
Move to EntryHi	Load/Store affected by new ASID	1
Move to EntryHi	Instruction fetch affected by new ASID	3
TLBP	Move from Coprocessor Zero Register	0
Move to Index Register	TLBR/TLBWI	1
Change to CU Bits in Status Register	Coprocessor Instruction	1
Move to EPC, ErrorPC or DEPC	ERET	1
Move to Status Register	ERET	0
Set of IP in Cause Register	Interrupted Instruction	3
Any Other Move to Coprocessor 0 Registers	Instruction Affected by Change	2
CACHE instruction operating on I$	Instruction fetch seeing new cache state	3
LL	Move From LLAddr	1

Table 11-1   Instruction Hazards (continued)

Instruction Hazards		
First Instruction	**Second Instruction**	**Spacing (Instructions)**
Move to Compare	Instruction not seeing TimerInterrupt*	4

11.3 CPU Opcode Map

Key

- CAPITALIZED text indicates an opcode mnemonic

- *Italicized* text indicates to look at the specified opcode submap for further instruction bit decode

- Entries containing the α symbol indicate that a reserved instruction fault occurs if the core executes this instruction.

- Entries containing the β symbol indicate that a coprocessor unusable exception occurs if the core executes this instruction

Table 11-2   CPU Main Opcode Map

Main Opcode Map		Opcode[28:26]							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Opcode [31:29]	0	*Special*	*RegImm*	J	JAL	BEQ	BNE	BLEZ	BGTZ
	1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
	2	*COP0*	β	β	β	BEQL	BNEL	BLEZL	BGTZL
	3	α	α	α	α	*Special2*	α	α	α
	4	LB	LH	LWL	LW	LBU	LHU	LWR	α
	5	SB	SH	SWL	SW	α	α	SWR	CACHE
	6	LL	β	β	PREF	α	β	β	α
	7	SC	β	β	α	α	β	β	α

Table 11-3   Special Submap

Special Submap		Opcode[2:0]							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Opcode [5:3]	0	SLL	β	SRL	SRA	SLLV	α	SRLV	SRAV
	1	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	α	SYNC
	2	MFHI	MTHI	MFLO	MTLO	α	α	α	α
	3	MULT	MUTLU	DIV	DIVU	α	α	α	α
	4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
	5	α	α	SLT	SLTU	α	α	α	α
	6	TGE	TGEU	TLT	TLTU	TEQ	α	TNE	α
	7	α	α	α	α	α	α	α	α

Table 11-4   Special2 Submap

Special2 Submap		Opcode[2:0]							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Opcode [5:3]	0	MADD	MADDU	MUL	α	MSUB	MSUBU	α	α
	1	α	α	α	α	α	α	α	α
	2	α	α	α	α	α	α	α	α
	3	α	α	α	α	α	α	α	α
	4	CLZ	CLO	α	α	α	α	α	α
	5	α	α	α	α	α	α	α	α
	6	α	α	α	α	α	α	α	α
	7	α	α	α	α	α	α	α	SDBBP

Table 11-5   Register Immediate Submap

RegImm Submap		Opcode[18:16]							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Opcode [20:19]	0	BLTZ	BGEZ	BLTZL	BGEZL	α	α	α	α
	1	TGEI	TGEIU	TLTI	TLTIU	TEQI	a	TNEI	α
	2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	α	α	α	α
	3	α	α	α	α	α	α	α	α

Table 11-6   Coprocessor 0 Rs Submap

COP0 Rs Submap		Opcode[23:21]							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Opcode [25:24]	0	MFCz	α	α	α	MTCz	α	α	α
	1	α	α	α	α	α	α	α	
	2	*COPz*							
	3								

Table 11-7   Coprocessor 0 Submap

COPZ Submap		Opcode[2:0]							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Opcode [5:3]	0	α	TLBR	TLBWI	α	α	α	TLBWR	α
	1	TLBP	α	α	α	α	α	α	α
	2	α	α	α	α	α	α	α	α
	3	ERET	α	α	α	α	α	α	DERET
	4	WAIT	α	α	α	α	α	α	α
	5	α	α	α	α	α	α	α	α
	6	α	α	α	α	α	α	α	α
	7	α	α	α	α	α	α	α	α

11.4 Instruction Set

This section describes the core instructions. Table 11-8 lists the instructions in alphabetical order, followed by a detailed description of each instruction.

Table 11-8   Instruction Set

Instruction	Description	Function
ADD	Integer Add	$Rd = Rs + Rt$
ADDI	Integer Add Immediate	$Rt = Rs + Immed$
ADDIU	Unsigned Integer Add Immediate	$Rt = Rs +_U Immed$
ADDU	Unsigned Integer Add	$Rd = Rs +_U Rt$

Table 11-8   Instruction Set (continued)

Instruction	Description	Function
AND	Logical AND	Rd = Rs & Rt
ANDI	Logical AND Immediate	Rt = Rs & (0_{16} ‖ Immed)
BEQ	Branch On Equal	if Rs == Rt PC += (int)offset

BEQL	Branch On Equal Likely	if Rs == Rt
		  PC += (int)offset
		else
		  Ignore Next Instruction

BGEZ	Branch on Greater Than or Equal To Zero	if !Rs[31] PC += (int)offset
BGEZAL	Branch on Greater Than or Equal To Zero And Link	GPR[31] = PC + 8 if !Rs[31] PC += (int)offset

BGEZALL	Branch on Greater Than or Equal To Zero And Link Likely	GPR[31] = PC + 8
		if !Rs[31]
		  PC += (int)offset
		else
		  Ignore Next Instruction

BGEZL	Branch on Greater Than or Equal To Zero Likely	if !Rs[31]
		  PC += (int)offset
		else
		  Ignore Next Instruction

BGTZ	Branch on Greater Than Zero	if !Rs[31] && Rs != 0 PC += (int)offset

BGTZL	Branch on Greater Than Zero Likely	if !Rs[31] && Rs != 0
		  PC += (int)offset
		else
		  Ignore Next Instruction

Table 11-8   Instruction Set (continued)

Instruction	Description	Function
BLEZ	Branch on Less Than or Equal to Zero	if Rs[31] || Rs == 0 PC += (int)offset
BLEZL	Branch on Less Than or Equal to Zero Likely	if Rs[31] || Rs == 0 PC += (int)offset else Ignore Next Instruction
BLTZ	Branch on Less Than Zero	if Rs[31] PC += (int)offset
BLTZAL	Branch on Less Than Zero And Link	GPR[31] = PC + 8 if Rs[31] PC += (int)offset
BLTZALL	Branch on Less Than Zero And Link Likely	GPR[31] = PC + 8 if Rs[31] PC += (int)offset else Ignore Next Instruction
BLTZL	Branch on Less Than Zero Likely	if Rs[31] PC += (int)offset else Ignore Next Instruction
BNE	Branch on Not Equal	if Rs != Rt PC += (int)offset
BNEL	Branch on Not Equal Likely	if Rs != Rt PC += (int)offset else Ignore Next Instruction
BREAK	Breakpoint	Break Exception
CACHE	Cache Operation	See Cache Description

Table 11-8   Instruction Set (continued)

Instruction	Description	Function
COP0	Coprocessor 0 Operation	See Coprocessor Description
CLO	Count Leading Ones	Rd = NumLeadingOnes(Rs)
CLZ	Count Leading Zeroes	Rd = NumLeadingZeroes(Rs)
DERET	Return from Debug Exception	PC = DEPC Exit Debug Mode
DIV	Divide	LO = (int)Rs / (int)Rt HI = (int)Rs % (int)Rt
DIVU	Unsigned Divide	LO = (uns)Rs / (uns)Rt HI = (uns)Rs % (uns)Rt
ERET	Return from Exception	if SR[2] PC = ErrorEPC else PC = EPC SR[1] = 0 SR[2] = 0 LL = 0
J	Unconditional Jump	PC = PC[31:28] || offset<<2
JAL	Jump and Link	GPR[31] = PC + 8 PC = PC[31:28] || offset<<2
JALR	Jump and Link Register	Rd = PC + 8 PC = Rs
JR	Jump Register	PC = Rs
LB	Load Byte	Rt = (byte)Mem[Rs+offset]
LBU	Unsigned Load Byte	Rt = (ubyte))Mem[Rs+offset]
LH	Load Halfword	Rt = (half)Mem[Rs+offset]
LHU	Unsigned Load Halfword	Rt = (uhalf)Mem[Rs+offset]

Table 11-8   Instruction Set (continued)

Instruction	Description	Function
LL	Load Linked Word	Rt = Mem[Rs+offset] LL = 1 LLAdr = Rs + offset
LUI	Load Upper Immediate	Rt = immediate << 16
LW	Load Word	Rt = Mem[Rs+offset]
LWL	Load Word Left	
LWR	Load Word Right	
MADD	Multiply-Add	HI, LO += (int)Rs * (int)Rt
MADDU	Multiply-Add Unsigned	HI, LO += (uns)Rs * (uns)Rt
MFC0	Move From Coprocessor 0	Rt = CPR[0, n, sel] = Rt
MFHI	Move From HI	Rd = HI
MFLO	Move From LO	Rd = LO
MOVN	Move Conditional on Not Zero	if GPR[rt] ≠ 0 then GPR[rd] ← GPR[rs]
MOVZ	Move Conditional on Zero	if GPR[rt] = 0 then GPR[rd] ← GPR[rs]
MSUB	Multiply-Subtract	HI, LO -= (int)Rs * (int)Rt
MSUBU	Multiply-Subtract Unsigned	HI, LO -= (uns)Rs * (uns)Rt
MTC0	Move To Coprocessor 0	CPR[0, n] = Rt SEL
MTHI	Move To HI	HI = Rs
MTLO	Move To LO	LO = Rs
MUL	Multiply with register write	HI | LO =Unpredictable Rd = LO
MULT	Integer Multiply	HI | LO = (int)Rs * (int)Rd
MULTU	Unsigned Multiply	HI | LO = (uns)Rs * (uns)Rd

Table 11-8   Instruction Set (continued)

Instruction	Description	Function
NOR	Logical NOR	Rd = ~(Rs | Rt)
OR	Logical OR	Rd = Rs | Rt
ORI	Logical OR Immediate	Rt = Rs | Immed
PREF	Prefetch	Load Specified Line into Cache
SB	Store Byte	(byte)Mem[Rs+offset] = Rt
SC	Store Conditional Word	if LL =1 mem[Rxoffs] = Rt Rt = LL
SDBBP	Software Debug Break Point	Trap to SW Debug Handler
SH	Store Half	(half)Mem[Rs+offset] = Rt
SLL	Shift Left Logical	Rd = Rt << sa
SLLV	Shift Left Logical Variable	Rd = Rt << Rs[4:0]

SLT	Set on Less Than	if (int)Rs < (int)Rt
		 Rd = 1
		else
		 Rd = 0

SLTI	Set on Less Than Immediate	if (int)Rs < (int)Immed
		 Rt = 1
		else
		 Rt = 0

SLTIU	Set on Less Than Immediate Unsigned	if (uns)Rs < (uns)Immed
		 Rt = 1
		else
		 Rt = 0

Table 11-8   Instruction Set (continued)

Instruction	Description	Function

SLTU	Set on Less Than Unsigned	if (uns)Rs < (uns)Immed
		 Rd = 1
		else
		 Rd = 0

SRA	Shift Right Arithmetic	Rd = (int)Rt >> sa
SRAV	Shift Right Arithmetic Variable	Rd = (int)Rt >> Rs[4:0]
SRL	Shift Right Logical	Rd = (uns)Rt >> sa
SRLV	Shift Right Logical Variable	Rd = (uns)Rt >> Rs[4:0]
SSNOP	Superscalar Inhibit No Operation	
SUB	Integer Subtract	Rt = (int)Rs - (int)Rd
SUBU	Unsigned Subtract	Rt = (uns)Rs - (uns)Rd
SW	Store Word	Mem[Rs+offset] = Rt
SWL	Store Word Left	
SWR	Store Word Right	
SYNC	Synchronize	
SYSCALL	System Call	SystemCallException
TEQ	Trap if Equal	if Rs == Rt TrapException
TEQI	Trap if Equal Immediate	if Rs == (int)Immed TrapException
TGE	Trap if Greater Than or Equal	if (int)Rs >= (int)Rt TrapException
TGEI	Trap if Greater Than or Equal Immediate	if (int)Rs >= (int)Immed TrapException
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	if (uns)Rs >= (uns)Immed TrapException

Table 11-8   Instruction Set (continued)

Instruction	Description	Function
TGEU	Trap if Greater Than or Equal Unsigned	if (uns)Rs >= (uns)Rt TrapException
TLBWI	Write Indexed TLB Entry	
TLBWR	Write Random TLB Entry	
TLBP	Probe TLB for Matching Entry	
TLBR	Read Index for TLB Entry	
TLT	Trap if Less Than	if (int)Rs < (int)Rt TrapException
TLTI	Trap if Less Than Immediate	if (int)Rs < (int)Immed TrapException
TLTIU	Trap if Less Than Immediate Unsigned	if (uns)Rs < (uns)Immed TrapException
TLTU	Trap if Less Than Unsigned	if (uns)Rs < (uns)Rt TrapException
TNE	Trap if Not Equal	if Rs != Rt TrapException
TNEI	Trap if Not Equal Immediate	if Rs != (int)Immed TrapException
WAIT	Wait for Interrupts	Stall until interrupt occurs
XOR	Exclusive OR	Rd = Rs ^ Rt
XORI	Exclusive OR Immediate	Rt = Rs ^ (uns)Immed

Add Word		**ADD**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000
6	5	5	5	5	6
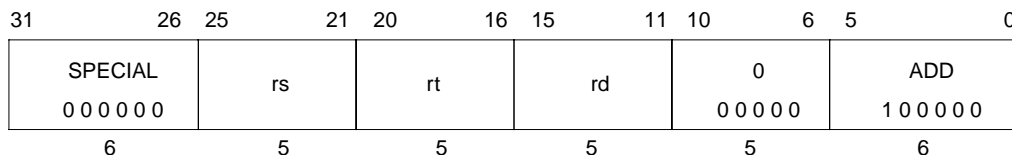
Format: ADD rd, rs, rt **MIPS I**

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

Description: rd ← rs + rt

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

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

- If the addition does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
temp ← (GPR[rs]_{31}||GPR[rs]_{31..0}) + (GPR[rt]_{31}||GPR[rt]_{31..0})
if temp_{32} ≠ temp_{31} then
SignalException(IntegerOverflow)
else
GPR[rd] ← sign_extend(temp_{31..0})
endif
```

Exceptions: Integer Overflow

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

| Add Immediate Word | ADDI |

31 26	25 21	20 16	15 0
ADDI 0 0 1 0 0 0	rs	rt	immediate
6	5	5	16
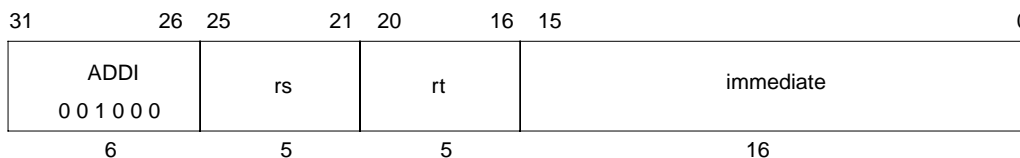
Format: ADDI   rt, rs, immediate   **MIPS I**

Purpose: To add a constant to a 32-bit integer. If overflow occurs, then trap**.**

Description: rd ← rs + immediate

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

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

- If the addition does not overflow, the 32-bit result is placed into GPR *rt*.

Restrictions:

None

Operation:

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

Exceptions: Integer Overflow

Programming Notes:

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

Add Immediate Unsigned Word   **ADDIU**

31 26	25 21	20 16	15 0
ADDIU 001001	rs	rt	immediate
6	5	5	16
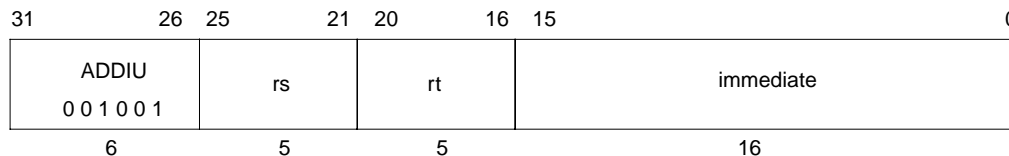
Format: ADDIU rt, rs, immediate   **MIPS I**

Purpose: To add a constant to a 32-bit integer

Description: rd ← rs + immediate

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

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt]← sign_extend(temp_{31..0})
```

Exceptions: None

Programming Notes:

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

Add Unsigned Word

ADDU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADDU 100001
6	5	5	5	5	6
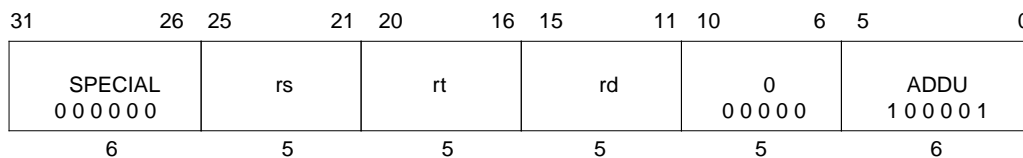
Format: ADDU   rd, rs, rt   **MIPS I**

Purpose: To add 32-bit integers

Description: rd ← rs + rt

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

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← sign_extend(temp₃₁..₀)
```

Exceptions: None

Programming Notes:

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

And | AND

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000		rs		rt		rd		0 00000		AND 100100	
6		5		5		5		5		6	
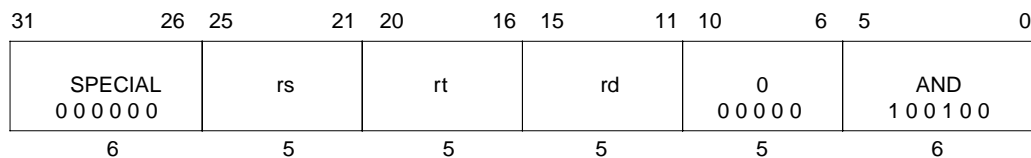
Format:　　　AND　　　rd, rs, rt　　　　　　　　　　　　　　　　**MIPS I**

Purpose: To do a bitwise logical AND

Description: rd ← rs AND rt

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

Restrictions: None

Operation:

GPR[rd] ← GPR[rs] and GPR[rt]

Exceptions: None

And Immediate   **ANDI**

31 26	25 21	20 16	15 0
ANDI 0 0 1 1 0 0	rs	rt	immediate
6	5	5	16
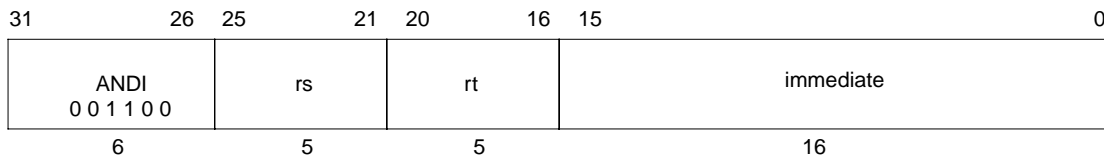
Format: ANDI   rt, rs, immediate   **MIPS I**

Purpose: To do a bitwise logical AND with a constant

Description: rt ← rs AND immediate

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

Restrictions: None

Operation:

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

Exceptions: None

Branch on Equal   **BEQ**

31 26	25 21	20 16	15 0
BEQ `0 0 0 1 0 0`	rs	rt	offset
6	5	5	16
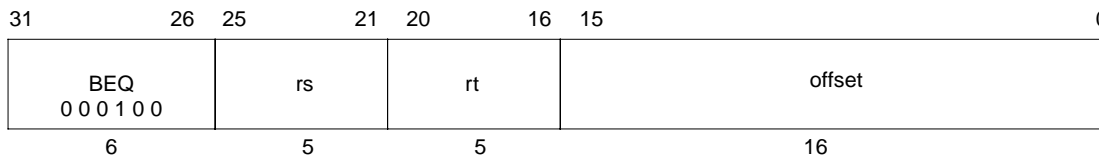
Format: BEQ   rs, rt, offset   **MIPS I**

Purpose: To compare GPRs then do a PC-relative conditional branch

Description: `if rs = rt then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions: None

Operation:

```
I:          target_offset ← sign_extend(offset || 0²)
            condition ← (GPR[rs] = GPR[rt])
            I+1:if condition then
            PC ← PC + target_offset
            endif
```

Exceptions: None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 Kbytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Branch on Equal Likely

31 26	25 21	20 16	15 0
BEQL 010100	rs	rt	offset
6	5	5	16
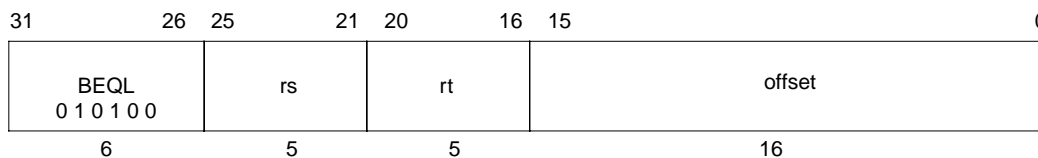
Format: BEQL   rs, rt, offset   **MIPS II**

Purpose: To compare GPRs then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: `if rs = rt then branch_likely`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions: None

Operation:
```
I:          target_offset ← sign_extend(offset || 0²)
            condition ← (GPR[rs] = GPR[rt])
            I+1:if condition then
            PC ← PC + target_offset
            else
            NullifyCurrentInstruction()
            endif
```

Programming Notes:

It is recommended that new software not use this instruction as it will be phased out in future versions of the MIPS32 architecture.

Branch on Greater Than or Equal to Zero BGEZ

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZ 00001	offset
6	5	5	16
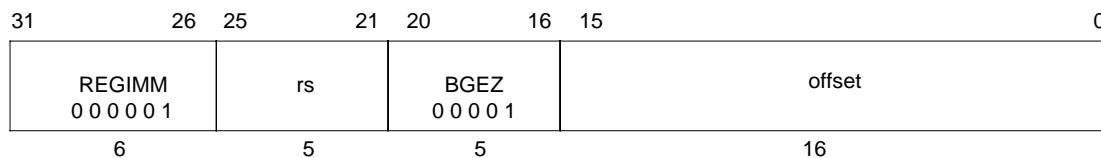
Format: BGEZ rs, offset **MIPS I**

Purpose: To test a GPR then do a PC-relative conditional branch

Description: if rs ≥ 0 then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions: None

Operation:

```
I:          target_offset ← sign_extend(offset || 0²)
            condition ← GPR[rs] ≥ 0^GPRLEN
            I+1:if condition then
            PC ← PC + target_offset
            endif
```

Exceptions: None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Branch on Greater Than or Equal to Zero and Link   **BGEZAL**

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	BGEZAL 1 0 0 0 1	offset
6	5	5	16
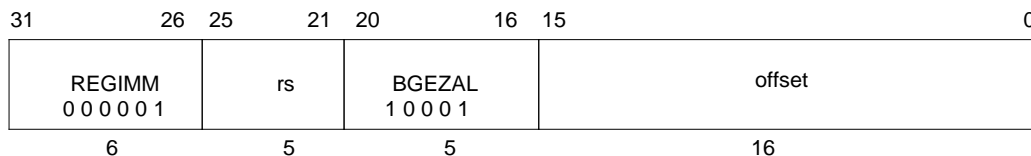
Format: BGEZAL   rs, offset   **MIPS I**

Purpose: To test a GPR then do a PC-relative conditional procedure call

Description: if rs $\geq$ 0 then procedure_call

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

GPR 31 must not be used for the source register *rs*, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Operation:

```
I:          target_offset ← sign_extend(offset || 0²)
            condition ← GPR[rs] ≥ 0^GPRLEN
            GPR[31] ← PC + 8
            I+1:if condition then
            PC ← PC + target_offset
            endif
```

Branch on Greater Than or Equal to Zero and Link (cont.)   **BGEZAL**

Exceptions: None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

Branch on Greater Than or Equal to Zero and Link Likely   **BGEZALL**

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	BGEZALL 1 0 0 1 1	offset
6	5	5	16
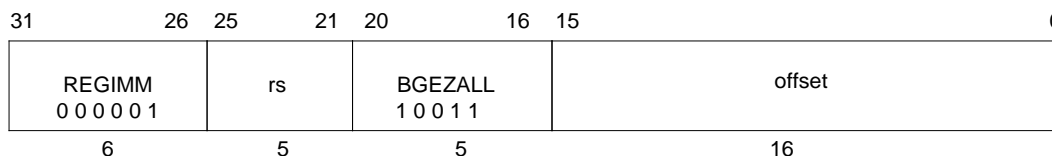
Format: BGEZALL   rs, offset   **MIPS II**

Purpose: To test a GPR then do a PC-relative conditional procedure call; execute the delay slot only if the branch is taken.

Description: if rs ≥ 0 then procedure_call_likely

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

GPR 31 must not be used for the source register *rs*, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Branch on Greater Than or Equal to Zero and Link Likely (cont.)	BGEZALL

Operation:

```
I:            target_offset ← sign_extend(offset || 0²)
              condition ← GPR[rs] ≥ 0^GPRLEN
              GPR[31] ← PC + 8
              I+1:if condition then
              PC ← PC + target_offset
              else
              NullifyCurrentInstruction()
              endif
```

I: $\text{target\_offset} \leftarrow \text{sign\_extend}(\text{offset} \ || \ 0^2)$
$\text{condition} \leftarrow \text{GPR[rs]} \geq 0^{GPRLEN}$
$\text{GPR[31]} \leftarrow \text{PC} + 8$

Exceptions: None

Programming Notes:

It is recommended that new software not use this instruction as it will be phased out in future versions of the MIPS32 architecture.

Branch on Greater Than or Equal to Zero Likely   BGEZL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZL 00011	offset
6	5	5	16
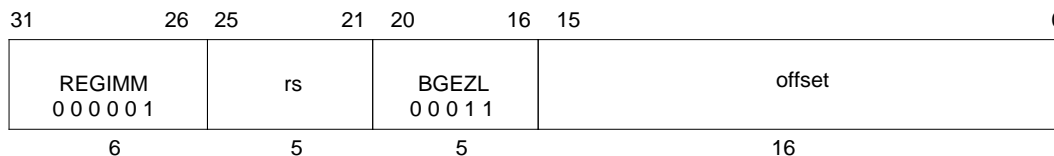
Format: BGEZL   rs, offset   **MIPS II**

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if rs ≥ 0 then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions: None

Operation:

```
I:          target_offset ← sign_extend(offset || 0²)
            condition ← GPR[rs] ≥ 0^GPRLEN
            I+1:if condition then
            PC ← PC + target_offset
            else
            NullifyCurrentInstruction()
            endif
```

Exceptions: None

Programming Notes:

It is recommended that new software not use this instruction as it will be phased out in future versions of the MIPS32 architecture.

Branch on Greater Than Zero   **BGTZ**

31 26	25 21	20 16	15 0
BGTZ 0 0 0 1 1 1	rs	0 0 0 0 0 0	offset
6	5	5	16

Format: BGTZ   rs, offset   **MIPS I**

Purpose: To test a GPR then do a PC-relative conditional branch

Description: if rs > 0 then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions: None

Operation:

```
I:          target_offset ← sign_extend(offset || 0²)
            condition ← GPR[rs] > 0^GPRLEN
            I+1:if condition then
            PC ← PC + target_offset
            endif
```

Exceptions: None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Branch on Greater Than Zero Likely

BGTZL

31 26	25 21	20 16	15 0
BGTZL 0 1 0 1 1 1	rs	0 0 0 0 0 0	offset
6	5	5	16

Format: BGTZL   rs, offset   **MIPS II**

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: `if rs > 0 then branch_likely`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions: None

Operation:

```
I:          target_offset ← sign_extend(offset || 0²)
            condition ← GPR[rs] > 0^GPRLEN
            I+1:if condition then
            PC ← PC + target_offset
            else
            NullifyCurrentInstruction()
            endif
```

Exceptions: None

Programming Notes:

It is recommended that new software not use this instruction as it will be phased out in future versions of the MIPS32 architecture.

| **Branch on Less Than or Equal to Zero** | | | | **BLEZ** |

31 26	25 21	20 16	15 0
BLEZ 0 0 0 1 1 0	rs	0 0 0 0 0 0	offset
6	5	5	16

Format: BLEZ   rs, offset   **MIPS I**

Purpose: To test a GPR then do a PC-relative conditional branch

Description: if rs ≤ 0 then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions: None

Operation:

```
I:              target_offset ← sign_extend(offset || 0²)
                condition ← GPR[rs] ≤ 0^GPRLEN
                I+1:if condition then
                PC ← PC + target_offset
                endif
```

Exceptions: None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Branch on Less Than or Equal to Zero Likely

BLEZL

31 26	25 21	20 16	15 0
BLEZL 0 1 0 1 1 0	rs	0 0 0 0 0 0	offset
6	5	5	16

Format: BLEZL   rs, offset   **MIPS II**

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if rs ≤ 0 then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions: None

Operation:

```
I:            target_offset ← sign_extend(offset || 0²)
              condition ← GPR[rs] ≤ 0^GPRLEN
              I+1:if condition then
              PC ← PC + target_offset
              else
              NullifyCurrentInstruction()
              endif
```

Exceptions: None

Programming Notes:

It is recommended that new software not use this instruction as it will be phased out in future versions of the MIPS32 architecture.

Branch on Less Than Zero			**BLTZ**

```
31              26 25          21 20        16 15                                    0
┌─────────────────┬─────────────┬─────────────┬──────────────────────────────────────┐
│     REGIMM      │             │    BLTZ     │                                      │
│                 │     rs      │             │               offset                │
│     000001      │             │   00000     │                                      │
└─────────────────┴─────────────┴─────────────┴──────────────────────────────────────┘
         6               5             5                        16
```

Format: BLTZ   rs, offset   **MIPS I**

Purpose: To test a GPR then do a PC-relative conditional branch

Description: `if rs < 0 then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions: None

Operation:

```
I:          target_offset ← sign_extend(offset || 0²)
            condition ← GPR[rs] < 0^GPRLEN
            I+1:if condition then
            PC ← PC + target_offset
            endif
```

Exceptions: None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

Branch on Less Than Zero and Link

BLTZAL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZAL 10000	offset
6	5	5	16
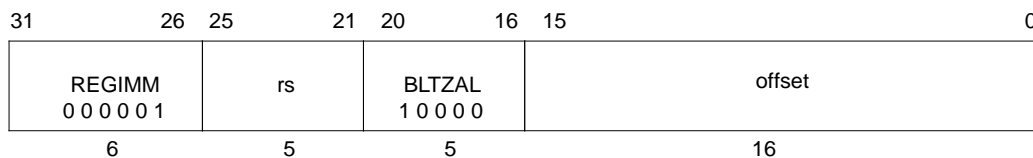
Format: BLTZAL rs, offset **MIPS I**

Purpose: To test a GPR then do a PC-relative conditional procedure call

Description: if rs < 0 then procedure_call

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

GPR 31 must not be used for the source register *rs*, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Branch on Less Than Zero and Link (cont.)　　　　　　　　　　　　　　　　　　　　**BLTZAL**

Operation:

I:　　　　　　target_offset ← sign_extend(offset || 0^2)
　　　　　　　condition ← GPR[rs] < 0^{GPRLEN}
　　　　　　　GPR[31] ← PC + 8
　　　　　　**I+1:**if condition then
　　　　　　PC ← PC + target_offset
　　　　　　endif

Exceptions: None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

Branch on Less Than Zero and Link Likely

BLTZALL

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	BLTZALL 1 0 0 1 0	offset
6	5	5	16
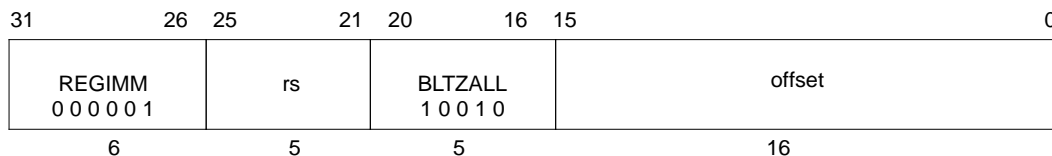
Format: BLTZALL rs, offset   **MIPS II**

Purpose: To test a GPR then do a PC-relative conditional procedure call; execute the delay slot only if the branch is taken.

Description: if rs < 0 then procedure_call_likely

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

GPR 31 must not be used for the source register *rs*, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

| Branch on Less Than Zero and Link Likely (cont.) | BLTZALL |

Operation:

```
I:              target_offset ← sign_extend(offset || 0²)
                condition ← GPR[rs] < 0^GPRLEN
                GPR[31] ← PC + 8
                I+1:if condition then
                PC ← PC + target_offset
                else
                NullifyCurrentInstruction()
                endif
```

Exceptions: Reserved Instruction

Programming Notes:

It is recommended that new software not use this instruction as it will be phased out in future versions of the MIPS32 architecture.

Branch on Less Than Zero Likely

BLTZL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZL 00010	offset
6	5	5	16
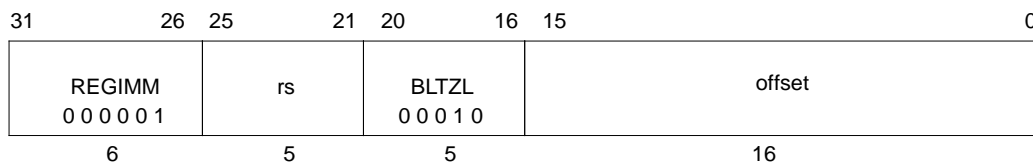
Format: BLTZL   rs, offset   **MIPS II**

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if rs < 0 then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions: None

Operation:

**I:**   target_offset ← sign_extend(offset || 0^2)
             condition ← GPR[rs] < 0$^{\text{GPRLEN}}$
             I+1:if condition then
             PC ← PC + target_offset
             else
             NullifyCurrentInstruction()
             endif

Exceptions: None

Programming Notes:

It is recommended that new software not use this instruction as it will be phased out in future versions of the MIPS32 architecture.

Branch on Not Equal   **BNE**

31 26	25 21	20 16	15 0
BNE 0 0 0 1 0 1	rs	rt	offset
6	5	5	16
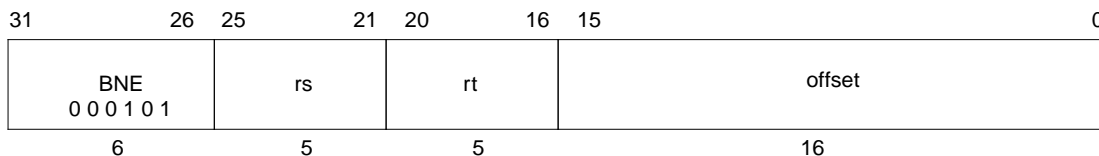
Format: BNE   rs, rt, offset   **MIPS I**

Purpose: To compare GPRs then do a PC-relative conditional branch

Description: `if rs ≠ rt then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are not equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions: None

Operation:

```
I:            target_offset ← sign_extend(offset || 0²)
              condition ← (GPR[rs] ≠ GPR[rt])
              I+1:if condition then
              PC ← PC + target_offset
              endif
```

Exceptions: None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Branch on Not Equal Likely   BNEL

31 26	25 21	20 16	15 0
BNEL 0 1 0 1 0 1	rs	rt	offset
6	5	5	16
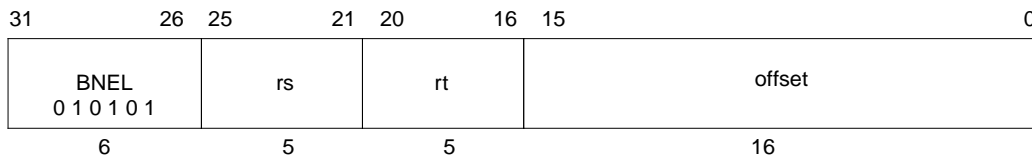
Format: BNEL   rs, rt, offset   **MIPS II**

Purpose: To compare GPRs then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: `if rs ≠ rt then branch_likely`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are not equal, branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions: None

Operation:

```
I:          target_offset ← sign_extend(offset || 0²)
            condition ← (GPR[rs] ≠ GPR[rt])
            I+1:if condition then
            PC ← PC + target_offset
            else
            NullifyCurrentInstruction()
            endif
```

Exceptions: None

Programming Notes:

It is recommended that new software not use this instruction as it will be phased out in future versions of the MIPS32 architecture.

Breakpoint — BREAK

31 26	25 6	5 0
SPECIAL 000000	code	BREAK 001101
6	20	6

Format: BREAK   **MIPS I**

Purpose: To cause a Breakpoint exception

Description:
A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.
The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading
the contents of the memory word containing the instruction.

Restrictions: None

Operation:

```
SignalException(Breakpoint)
```

Exceptions: Breakpoint

Perform Cache Operation	CACHE

31	26	25	21	20	16	15	0
CACHE 1 0 1 1 1 1		base		op		Offset	
6		5		5		16	
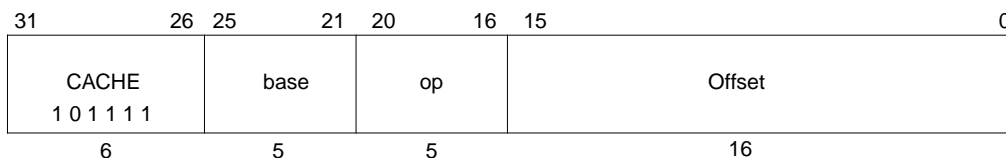
Format:

```
CACHE         op, offset(base)                                    MIPS32
```

Purpose:

To perform the cache operation specified by op.

Description:

The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of three ways based on the operation to be performed and the type of cache as described in the following table.

Operation Requires an	Type of Cache	Usage of Effective Address
Address	Physical	The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache
Index	N/A	The effective address is translated by the MMU to a physical address. The address is used to index the cache. Assuming that the total cache size in bytes is CS, the associativity is A, and the number of bytes per tag is BPT, the following calculations give the fields of the address which specify the way and the index: $$OffsetBit <- Log2(BPT)$$ $$IndexBit <- Log2(CS / A)$$ $$WayBit <- IndexBit + Ceiling(Log2(A))$$ $$Way <- Addr_{WayBit-1..IndexBit}$$ $$Index <- Addr_{IndexBit-1..OffsetBit}$$ For a direct-mapped cache, the Way calculation is ignored and the Index value fully specifies the cache tag. This is shown symbolically in Figure 11-3.
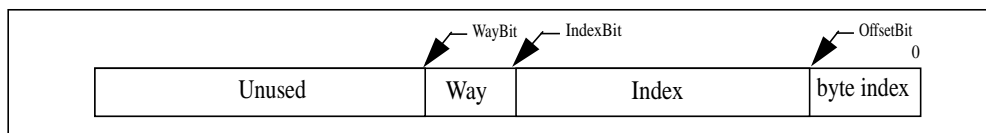
Figure 11-3 Usage of Address Fields to Select Index and Way

A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur on any operation. For index operations (where the address is used to index the cache but need not match the cache tag) unmapped addresses may be used to avoid TLB exceptions. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS nor data Watch exceptions.

Bits [17:16] of the instruction specify the cache on which to perform the operation, as follows:

Table 11-9   Encoding of CACHE Instruction Bits[17:16]

Code	Name	Cache
0 0	I	Primary Instruction
0 1	D	Primary Data or Unified Primary
1 0	Reserved	Not supported on 4K cores
1 1	Reserved	Not supported on 4K cores

Bits [20:18] of the instruction specify the operation to perform. On Index Load Tag operations, the specific word that is addressed in loaded into the DataLo register. All other **CACHE** instructions are line based and the word and byte indexes will not affect their operation..

Table 11-10 Encoding of CACHE Instruction Bits [20:18]

Code	Caches	Name	Effective Address Operand Type	Operation
0 0 0	I,D	Index Invalidate	Index	Set the state of the cache block at the specified index to invalid.
0 0 1	I, D	Index Load Tag	Index	Read the tag for the cache block at the specified index into the TagLo COP0 register. Also read the wordcorresponding to the byte index into the DataLo register.

Table 11-10 Encoding of CACHE Instruction Bits [20:18] (continued)

Code	Caches	Name	Effective Address Operand Type	Operation
0 1 0	I, D	Index Store Tag	Index	Write the tag for the cache block at the specified index from the TagLo and TagHi COP0 registers.
0 1 1		Reserved		Treated as a NOP.
1 0 0	I, D	Hit Invalidate	Address	If the cache block contains the specified address, set the state of the cache block to invalid.
1 0 1	I	Fill	Address	Fill the cache from the specified address. The cache line will be re-fetched even if it is already in the cache.
	D	Hit Invalidate	Address	For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid.
1 1 0	D	Hit Writeback	Address	This operation is treated as a NOP.
1 1 1	I, D	Fetch and Lock	Address.	If the cache does not contain the entire line at the specified address it is fetched from memory, and the state is set to locked. If the cache already contains the line, set the state to locked. The lock state may be cleared by executing an Index Invalidate or Hit Invalidate operation to the locked line, or via an Index Store Tag operation to the line that clears the lock bit.

Restrictions:

Execution of this instruction is legal only if the processor is operating in kernel mode, or if the CP0 enable bit is set in the Status register. In other circumstances, a Coprocessor Unusable Exception is taken.

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

Operation:

```
if (SR_CU0 = 1) or (SR_UM = 0) or (SR_EXL = 1) or (SR_ERL = 1) then
```

```
                          vAddr <- GPR[base] + sign_extend(offset)
                          (pAddr, uncached) <- AddressTranslation(vAddr, DataReadReference)
                          CacheOp(op, vAddr, pAddr)
                     else
                          InitiateCoprocessorUnusableException(0)
                     endif
```

Exceptions:
TLB Refill Exception.
TLB Invalid Exception
Coprocessor Unusable Exception

Count Leading Ones in Word					CLO

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 0 1 1 1 0 0	rs	0 0 0 0 0 0	rd	0 0 0 0 0 0	CLO 1 0 0 0 0 1
6	5	5	5	5	6
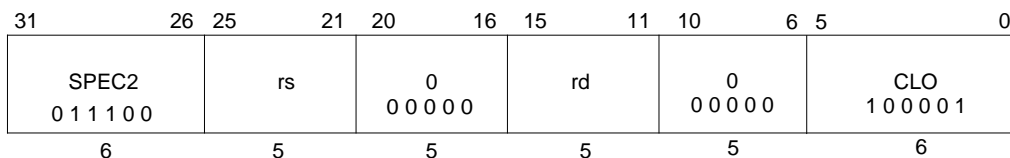
Format:

CLO   rd, rs   MIPS32

Purpose:

Count the number of leading ones in a word

Description:

The 32-bit word in GPR *rs* is scanned from most significant to least significant bit. The number of leading ones is counted and the result is written to GPR *rd* If all 32 bits were set in GPR *rs*, the result written to GPR *rd* is 32.

Restrictions:

None

Operation:

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

Exceptions:

None

31　　　　　26	25　　　　21	20　　　　16	15　　　　11	10　　　6	5　　　　　0

SPEC2 0 1 1 1 0 0	rs	0 0 0 0 0 0	rd	0 0 0 0 0 0	CLZ 1 0 0 0 0 0
6	5	5	5	5	6

Count Leading Zeros in Word　　　　　　　　　　　　　　　　　　　　　　　**CLZ**
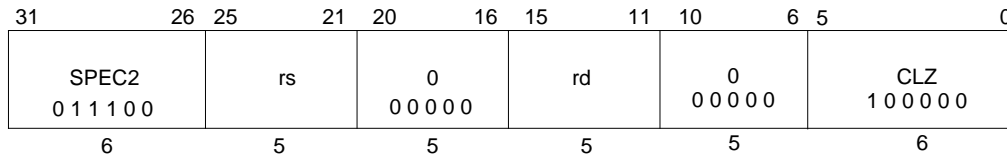
Format:

    CLZ   rd, rs   MIPS32

Purpose

 Count the number of leading zeros in a word

Description:

The 32-bit word in GPR *rs* is scanned from most significant to least significant bit. The number of leading zeros is counted and the result is written to GPR *rd*. If no bits were set in GPR *rs*, the result written to GPR *rd* is 32.

Restrictions:

None

Operation:

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

Exceptions:

None

Coprocessor Operation for Coprocessor 0 — COP0

31 26	25 24	0
COP0 010000	CO 1	Coprocessor Function
6	1	25
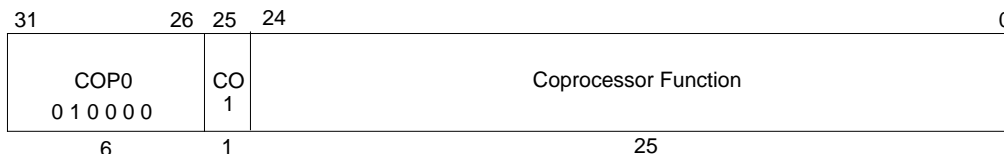
Format:
```
COP0   rt, rd                                              MIPS32
```

Purpose:

Perform the coprocessor function specified by Bits [24:0].

Description:

A coprocessor function, as described by Bits [24:0] is performed that is specific to coprocessor *0*. Refer to the instruction descriptions for each coprocessor for more details.

Restrictions:

If the coprocessor enable bit for coprocessor *0* is off in the Status register, execution of this instruction results in a Coprocessor Unusable Exception. For coprocessor 0, this instruction is legal only if the processor is in kernel mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

Operation:

if $(SR_{CU0} = 1)$ or
   $((SR_{UM} = 0)$ or $(SR_{EXL} = 1)$ or $(SR_{ERL} = 1))$ then
        CoprocessorOperation(z, CoprocessorFunction)
else
        InitiateCoprocessorUnusableException (0)
endif

Exceptions:

Coprocessor Unusable Exception (if access is not allowed)
Reserved Instruction Exception (if access is allowed, but function not implemented)

Debug Exception Return   **DERET**

31 26	25 24	6	5 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DERET 0 1 1 1 1 1
6	1	19	6

Format:
    DERET   MIPS32

Purpose:
    Return from debug exception.

Description:
    DERET returns to normal mode at the instruction pointed to by DEPC, e.g. the instruction that received the debug exception. DERET does not execute the next instruction (i.e. it has no delay slot).

Restrictions:
    The operation of the processor is **UNDEFINED** if a DERET is placed in the delay slot of a branch or jump instruction. A DERET placed between an LL and SC instruction does not cause the SC to fail. This instruction is legal only if the processor is in kernel or debug mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a coprocessor unusable exception.

    If the DEPC register with the return address was modified by an MTC0 instruction, then a minimum of two instructions must be executed before executing the DERET. The DERET instruction implements a software barrier for all changes in the CP0 state that could affect the fetch and decode of the instruction at the PC to which the DERET returns, such as changes to the effective ASID, user-mode state, and addressing mode.

Operation:

```
if (SR_CU0 = 1) or (SR_UM = 0) or (SR_EXL = 1) or (SR_ERL = 1) or (Debug_DM =
1) then
Debug_DM <- 0
PC <- DEPC
else
SignalException(CoprocessorUnusable)
endif
```

Exceptions:
    Coprocessor Unusable Exception

Divide Word

DIV

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DIV 0 1 1 0 1 0
6	5	5	10	6

Format: DIV   rs, rt   **MIPS I**

Purpose: To divide 32-bit signed integers

Description: (LO, HI) ← rs / rt

The 32-bit word value in GPR *rs* is divided by the 32-bit value in GPR *rt,* treating both operands as signed values. The 32-bit quotient is placed into special register *LO* and the 32-bit remainder is placed into special register *HI*. No arithmetic exception occurs under any circumstances.

Restrictions:

None

If the divisor in GPR *rt* is zero, the arithmetic result value is undefined.

Divide Word (cont.)	DIV

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
        UndefinedResult()
        endif
```

MIPS I-III

$\text{I:}q \leftarrow GPR[rs]_{31..0} \text{ div } GPR[rt]_{31..0}$
$LO \leftarrow sign\_extend(q_{31..0})$
$r \leftarrow GPR[rs]_{31..0} \text{ mod } GPR[rt]_{31..0}$
$HI \leftarrow sign\_extend(r_{31..0})$

Exceptions: None

Programming Notes:

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

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

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

Divide Unsigned Word

DIVU

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DIVU 0 1 1 0 1 1
6	5	5	10	6
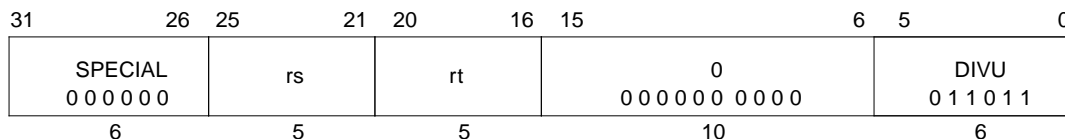
Format: DIVU   rs, rt   **MIPS I**

Purpose: To divide 32-bit unsigned integers

Description: (LO, HI) ← rs / rt

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

No arithmetic exception occurs under any circumstances.

Restrictions:

If the divisor in GPR *rt* is zero, the arithmetic result value is undefined.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
        UndefinedResult()
        endif
        I:q← (0 || GPR[rs]₃₁..₀) div (0 || GPR[rt]₃₁..₀)
        r← (0 || GPR[rs]₃₁..₀) mod (0 || GPR[rt]₃₁..₀)
        LO← sign_extend(q₃₁..₀)
        HI← sign_extend(r₃₁..₀)
```

Exceptions: None

Programming Notes: See "Programming Notes" for the DIV instruction.

Exception Return   ERET

```
31              26  25   24                                              0
┌──────────────┬──────┬────────────────────────────────┬──────────────┐
│    COP0      │  CO  │              0                 │    ERET      │
│   010000     │  1   │   000 0000 0000 0000 0000      │   011000     │
└──────────────┴──────┴────────────────────────────────┴──────────────┘
      6            1                 19                        6
```
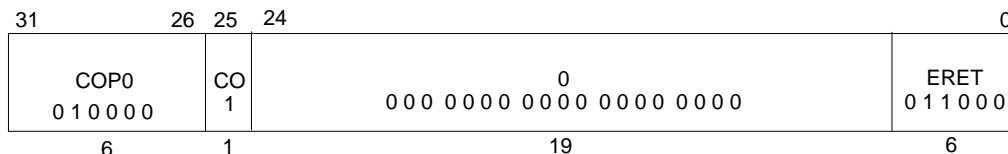
Format:
    ERET   MIPS32

Purpose:
    Return from interrupt, exception, or error trap

Description:
    **ERET returns to the interrupted instruction at the completion of interrupt, exception, or error trap process-
ing. ERET does not execute the next instruction (i.e., it has no delay slot).**

Restrictions:
    The operation of the processor is **UNDEFINED** if an ERET is placed in the delay slot of a branch or jump instruc-
tion.

    An ERET placed between an LL and SC instruction will always cause the SC to fail.

    This instruction is legal only if the processor is in kernel mode, or if the CP0 usable bit is set in the Status register. In
other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

    ERET implements a software barrier for all changes in the CP0 state that could affect the fetch and decode of the
instruction at the PC to which the ERET returns, such as changes to the effective ASID, user-mode state, and address-
ing mode.

Operation:
```
if (SR_CU0 = 1) or (SR_UM = 0) or (SR_EXL = 1) or (SR_ERL = 1) then
if SR_ERL = 1 then
PC <- ErrorEPC
SR_ERL <- 0
else
PC <- EPC
SR_EXL <- 0
endif
```

```
LLbit <- 0
else
InitiateCoprocessorUnusableException(0)
endif
```

Exceptions:

Coprocessor Unusable Exception

Jump		**J**

```
31          26  25                                              0
```

J 0 0 0 0 1 0	instr_index
6	26

Format: J   target   **MIPS I**

Purpose: To branch within the current 256 MB-aligned region

Description:

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

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

Restrictions: None

Exceptions: None

Operation:

I:

$\quad\quad$ **I+1:** $PC \leftarrow PC_{\text{GPRLEN}..28} \; || \; \text{instr\_index} \; || \; 0^2$

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset. This definition creates the following boundary case: When the branch instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.

Jump and Link JAL

```
31          26  25                                                    0
┌──────────────┬────────────────────────────────────────────────────┐
│     JAL      │                    instr_index                     │
│   000011     │                                                    │
└──────────────┴────────────────────────────────────────────────────┘
       6                                26
```

Format: JAL   target   **MIPS I**

Purpose: To execute a procedure call within the current 256 MB-aligned region

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call.

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

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

Restrictions: None

Operation:

```
I:              GPR[31]                    ← PC + 8
        I+1:PC ← PC_GPRLEN..28 || instr_index || 0²
```

Exceptions: None

Jump and Link (cont.)	JAL

Programming Notes:

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

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

Jump and Link Register

JALR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	0 0 0 0 0	rd	hint	JALR 0 0 1 0 0 1
6	5	5	5	5	6

Format: JALR   rs (rd = 31 implied)   **MIPS I**
              JALR   rd, rs   **MIPS I**

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

Description: rd ← return_addr, PC ← rs

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

Jump to the effective target address in GPR *rs*. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

At this time the only defined hint field value is 0, which sets default handling of JALR. Future implementations may define additional hint values.

Restrictions:

Register specifiers *rs* and *rd* must not be equal, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

The effective target address in GPR *rs* must be naturally-aligned. If either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction.

Jump and Link Register (cont.)	JALR

Operation:

```
I:         temp                        ← GPR[rs]
           GPR[rd] ← PC + 8
       I+1:PC ← temp
```

Exceptions: None

Programming Notes:

This is the only branch-and-link instruction that can select a register for the return link; all other link instructions use GPR 31. The default register for GPR *rd*, if omitted in the assembly language instruction, is GPR 31.

Jump Register   JR

31	26	25	21	20	11	10	6	5	0
SPECIAL 000000		rs		00 0000 0000		hint		JR 001000	
6		5		10		5		6	
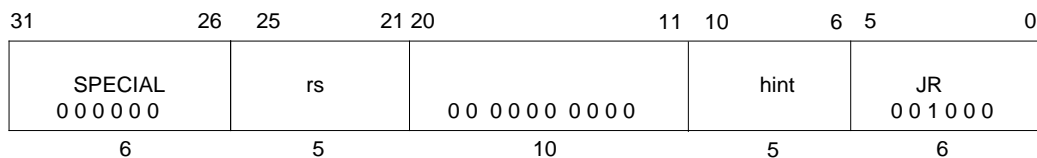
Format: JR   rs   **MIPS I**

Purpose: To execute a branch to an instruction address in a register

Description: PC ← rs

Jump to the effective target address in GPR *rs*. Execute the instruction following the jump, in the branch delay slot, before jumping.

Restrictions:

The effective target address in GPR *rs* must be naturally-aligned. If either of the 2 least-significant bits are not zero, then an Address Error exception occurs when the branch target is subsequently fetched as an instruction.

At this time the only defined hint field value is 0, which sets default handling of JR. Future implementations may define additional hint values.

Operation:

```
I:          temp         ← GPR[rs]
       I+1:PC ← temp
```

Exceptions: None

Programming Notes:

Software should use the value 31 for the *rs* field of the instruction word on return from a JAL, JALR, or BGEZAL, and should use a value other than 31 for remaining uses of JR.

Load Byte

LB

31		26	25		21	20		16	15		0

LB 1 0 0 0 0 0	base	rt	offset
6	5	5	16

Format: LB   rt, offset(base)   **MIPS I**

Purpose: To load a byte from memory as a signed value

Description: rt ← memory[base+offset]

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions: None

Operation:

```
vAddr                      ← sign_extend(offset) + GPR[base]
            (pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
            pAddr← pAddr_{PSIZE-1..2} || (pAddr_{1..0} xor ReverseEndian^2)
            memword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
            byte← vAddr_{1..0} xor BigEndianCPU^2
            GPR[rt]← sign_extend(memword_{7+8*byte..8*byte})
```

Exceptions: TLB Refill, TLB Invalid, Address Error

Load Byte Unsigned · LBU

31　　　　　26	25　　　　21	20　　　　16	15　　　　　　　　　　　　　　0
LBU 1 0 0 1 0 0	base	rt	offset
6	5	5	16

Format:　　　　　LBU　　　　　　rt, offset(base)　　　**MIPS I**

Purpose: To load a byte from memory as an unsigned value

Description: rt ← memory[base+offset]

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions: None

Operation:

```
vAddr                       ← sign_extend(offset) + GPR[base]
            (pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
            pAddr← pAddr_{PSIZE-1..2} || (pAddr_{1..0} xor ReverseEndian²)
            memword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
            byte← vAddr_{1..0} xor BigEndianCPU²
            GPR[rt] ← zero_extend(memword_{7+8* byte..8* byte})
```

Exceptions: TLB Refill, TLB Invalid, Address Error

Load Halfword   LH

31 26	25 21	20 16	15 0
LH 1 0 0 0 0 1	base	rt	offset
6	5	5	16
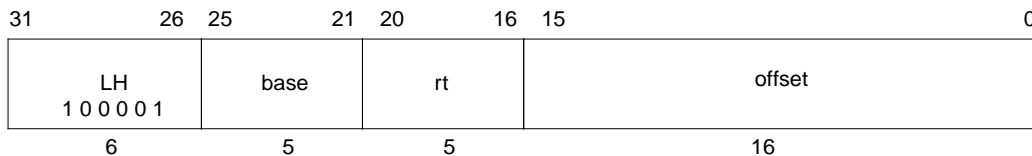
Format: LH   rt, offset(base)   **MIPS I**

Purpose: To load a halfword from memory as a signed value

Description: rt ← memory[base+offset]

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

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

Operation:

```
vAddr                      ← sign_extend(offset) + GPR[base]
          if vAddr₀ ≠ 0 then SignalException(AddressError) endif
          (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
          pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor (ReverseEndian || 0))
          memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
          byte ← vAddr_1..0 xor (BigEndianCPU || 0)
          GPR[rt] ← sign_extend(memword_15+8*byte..8* byte)
```

Exceptions: TLB Refill, TLB Invalid, Bus Error, Address Error

Load Halfword Unsigned

<div align="right">**LHU**</div>

31 26	25 21	20 16	15 0
LHU 1 0 0 1 0 1	base	rt	offset
6	5	5	16
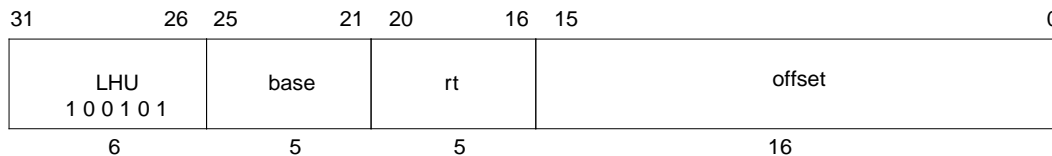
Format: LHU   rt, offset(base)   **MIPS I**

Purpose: To load a halfword from memory as an unsigned value

Description: rt ← memory[base+offset]

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

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

Operation:

```
vAddr                     ← sign_extend(offset) + GPR[base]
        if vAddr₀ ≠ 0 then SignalException(AddressError) endif
        hwsel ← (vAddr₁ xor BigEndianCPU) || 0
        vAddr ← vAddr_PSIZE-1..2 || hwsel
        memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
        (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
        memword← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
        GPR[rt] ← zero_extend(memword₁₅₊₍₈*hwsel₎..₈*hwsel)
```

Exceptions: TLB Refill, TLB Invalid, Address Error

Load Linked Word

<div align="right">

LL

</div>

31 26	25 21	20 16	15 0
LL 1 1 0 0 0 0	base	rt	offset
6	5	5	16
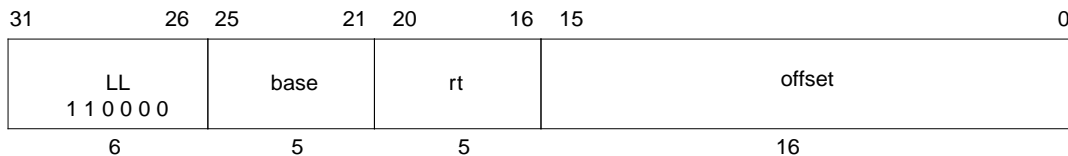
Format: LL   rt, offset(base)   **MIPS II**

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

Description: rt ← memory[base+offset]

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

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address. The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and written into GPR *rt*.

This begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor.

When an LL is executed it starts an active RMW sequence replacing any other sequence that was active.

The RMW sequence is completed by a subsequent SC instruction that either completes the RMW sequence atomically and succeeds, or does not and fails.

Executing LL on one processor does not cause an action that, by itself, causes an SC for the same block to fail on another processor.

An execution of LL does not have to be followed by execution of SC; a program is free to abandon the RMW sequence without attempting a write.

Load Linked Word (cont.)   **LL**

Restrictions:

The addressed location must be cached; if it is not, the result is undefined.

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

Operation:

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

Exceptions: TLB Refill, TLB Invalid, Address Error, Reserved Instruction

Programming Notes:

There is no Load Linked Word Unsigned operation corresponding to Load Word Unsigned.

Load Upper Immediate — LUI

31 26	25 21	20 16	15 0
LUI 0 0 1 1 1 1	0 0 0 0 0 0	rt	immediate
6	5	5	16
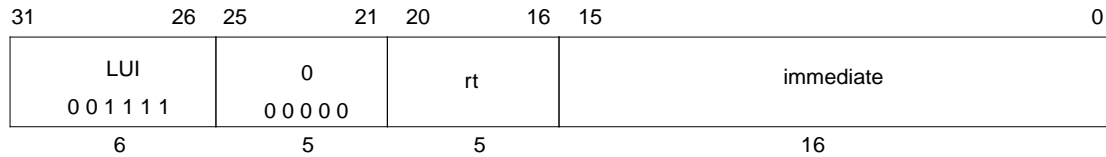
Format: LUI rt, immediate **MIPS I**

Purpose: To load a constant into the upper half of a word

Description: rt ← immediate || 0^{16}

The 16-bit *immediate* is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is sign-extended and placed into GPR *rt*.

Restrictions: None

Operation:

GPR[rt] ← sign_extend(immediate || 0^{16})

Exceptions: None

Load Word

LW

31	26	25	21	20	16	15	0
LW 1 0 0 0 1 1		base		rt		offset	
6		5		5		16	
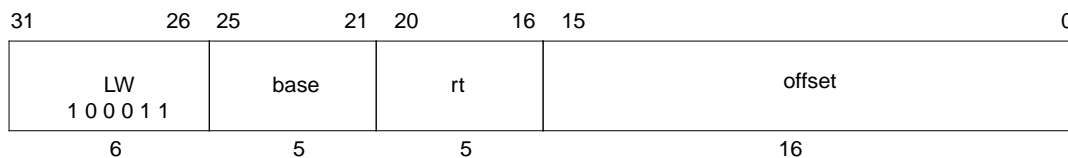
Format: LW   rt, offset(base)   **MIPS I**

Purpose: To load a word from memory as a signed value

Description: rt ← memory[base+offset]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

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

Operation:

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

Exceptions: TLB Refill, TLB Invalid, Bus Error, Address Error

Load Word Left   LWL

LWL 1 0 0 0 1 0	base	rt	offset
6	5	5	16

31   26 25   21 20   16 15   0
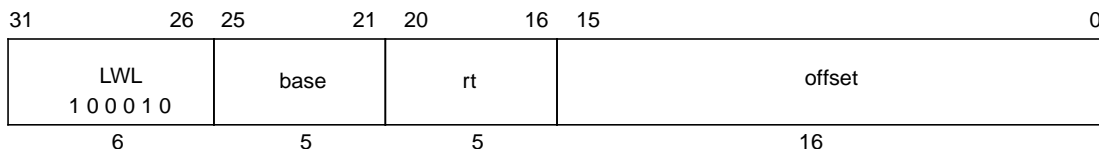
Format: LWL   rt, offset(base)   **MIPS I**

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

Description: rt ← rt MERGE memory[base+offset]

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr)*. *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W)* in memory starting at an arbitrary byte boundary.

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

Figure 11-4 illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2 bytes into the left part of the destination register word and leaves the right part of the destination word unchanged. Next, the complementary LWR loads the remainder of the unaligned word
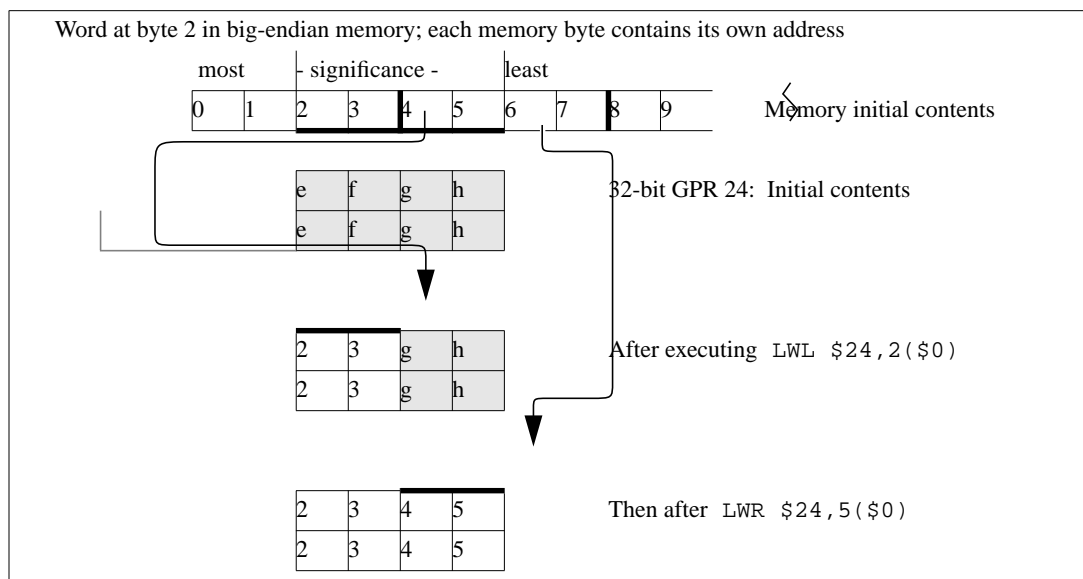
Figure 11-4 Unaligned Word Load Using LWL and LWR

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address ($vAddr_{1..0}$), and the current byte-ordering mode of the processor (big- or little-endian). Figure 11-5 shows the bytes loaded for every combination of offset and byte ordering.

Load Word Left (cont.)	LWL
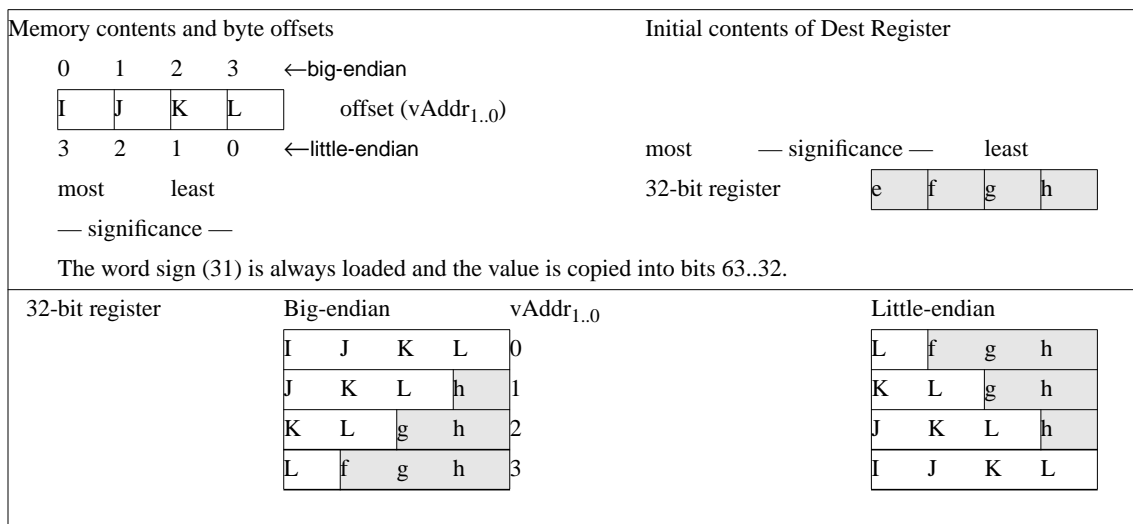
Figure 11-5 Bytes Loaded by LWL Instruction

The unaligned loads, LWL and LWR, are exceptions to the load-delay scheduling restriction in MIPS I architecture (see *Restrictions* below). An unaligned load instruction to GPR *rt* that immediately follows another load to GPR *rt* can read the loaded data. It correctly merges the 1 to 4 loaded bytes with the data loaded by the previous instruction.

Load Word Left (cont.)	LWL

Restrictions: None

Operation:

```
vAddr                    ← sign_extend(offset) + GPR[base]
         (pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
         pAddr← pAddr_{PSIZE-1..2} || (pAddr_{1..0} xor ReverseEndian²)
         if BigEndianMem = 0 then
         pAddr← pAddr_{PSIZE-1..2} || 0²
         endif
         byte← vAddr_{1..0} xor BigEndianCPU²
         memword← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
         GPR[rt]← memword_{7+8*byte..0} || GPR[rt]_{23-8*byte..0}
```

Exceptions: TLB Refill, TLB Invalid, Bus Error, Address Error

Load Word Right LWR

31 26	25 21	20 16	15 0
LWR 100110	base	rt	offset
6	5	5	16
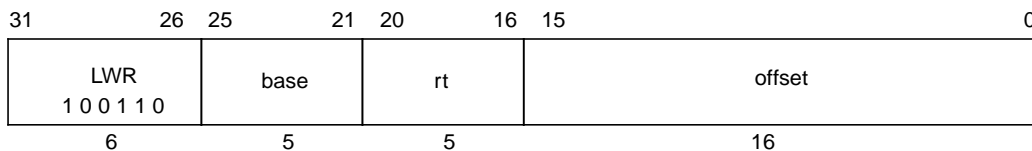
Format: LWR   rt, offset(base)   **MIPS I**

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

Description: rt ← rt MERGE memory[base+offset]

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr)*. *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word *(W)* in memory starting at an arbitrary byte boundary.

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

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

Figure 11-6 illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the least-significant byte at 5. First, LWR loads these 2 bytes into the right part of the destination register. Next, the complementary LWL loads the remainder of the unaligned word.

Load Word Right (cont.)
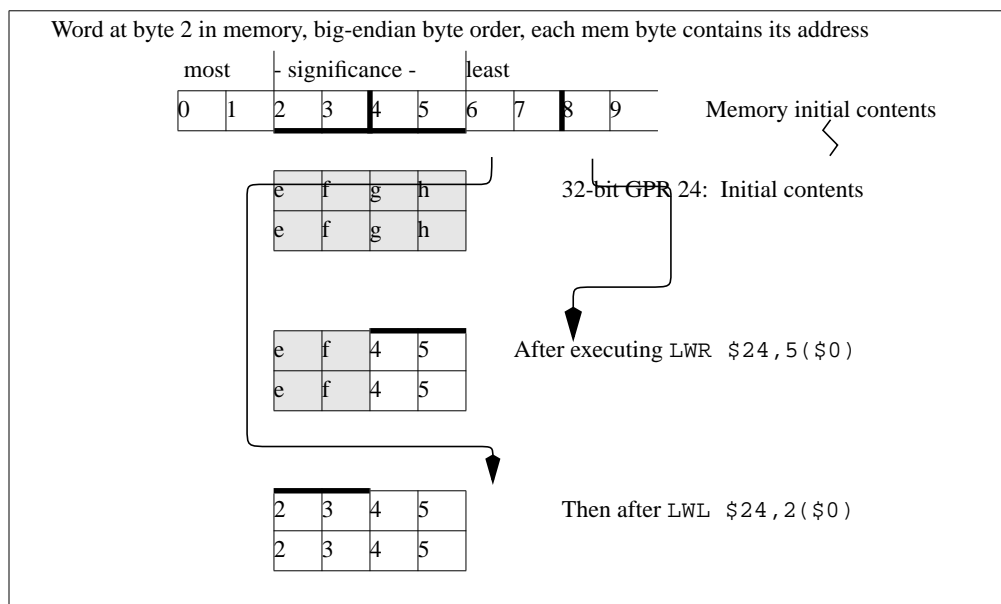
Figure 11-6 Unaligned Word Load Using LWR and LWL

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address ($vAddr_{1..0}$)—and the current byte-ordering mode of the processor (big- or little-endian). Figure 11-7 shows the bytes loaded for every combination of offset and byte ordering.

Load Word Right (cont.)	LWR

Memory contents and byte offsets

Initial contents of Dest Register

| 0 | 1 | 2 | 3 | ←big-endian |

| I | J | K | L | offset (vAddr1..0) |

| 3 | 2 | 1 | 0 | ←little-endian |

most   least

— significance —

most   — significance —   least

32-bit register | e | f | g | h |

32-bit register   big-endian   vAddr1..0   little-endian

e	f	g	I	0
e	f	I	J	1
e	I	J	K	2
I	J	K	L	3

I	J	K	L
e	I	J	K
e	f	I	J
e	f	g	I
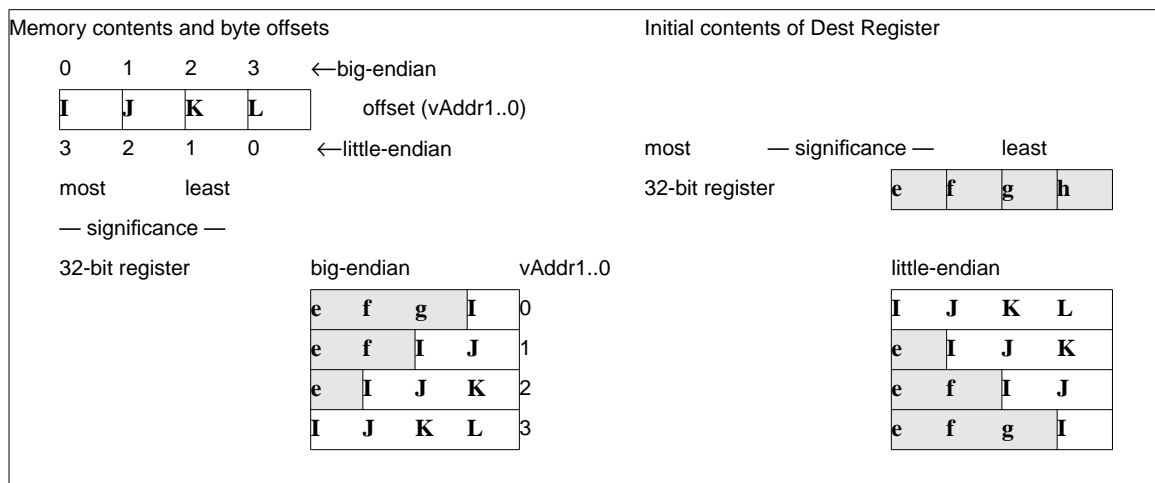
Figure 11-7 Bytes Loaded by LWR Instruction

The unaligned loads, LWL and LWR, are exceptions to the load-delay scheduling restriction in the MIPS I architecture. An unaligned load to GPR *rt* that immediately follows another load to GPR *rt* can "read" the loaded data. It correctly merges the 1 to 4 loaded bytes with the data loaded by the previous instruction.

Load Word Right (cont.) **LWR**

Restrictions: None

Operation:

```
vAddr                    ← sign_extend(offset) + GPR[base]
         (pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
         pAddr← pAddrₚSIZE-1..2 || (pAddr₁..₀ xor ReverseEndian²)
         if BigEndianMem = 0 then
         pAddr← pAddrₚSIZE-1..2 || 0²
         endif
         byte← vAddr₁..₀ xor BigEndianCPU²
         memword← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
         GPR[rt]← memword₃₁..₃₂-₈*byte || GPR[rt]₃₁-₈*byte..₀
```

$$vAddr \leftarrow \text{sign\_extend(offset)} + GPR[base]$$
$$(pAddr, CCA) \leftarrow AddressTranslation\ (vAddr, DATA, LOAD)$$
$$pAddr \leftarrow pAddr_{PSIZE-1..2}\ ||\ (pAddr_{1..0}\ xor\ ReverseEndian^2)$$
$$\text{if BigEndianMem = 0 then}$$
$$pAddr \leftarrow pAddr_{PSIZE-1..2}\ ||\ 0^2$$
$$\text{endif}$$
$$byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$$
$$memword \leftarrow LoadMemory\ (CCA, byte, pAddr, vAddr, DATA)$$
$$GPR[rt] \leftarrow memword_{31..32-8*byte}\ ||\ GPR[rt]_{31-8*byte..0}$$

Exceptions: TLB Refill, TLB Invalid, Bus Error, Address Error

Multiply and Add Word to Hi,Lo   **MADD**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 0 1 1 1 0 0	rs	rt	0 0 0 0 0 0	0 0 0 0 0 0	MADD 0 0 0 0 0 0
6	5	5	5	5	6
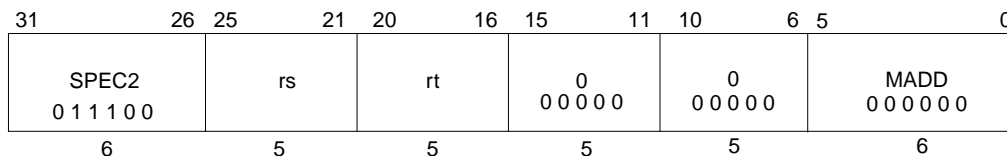
Format:

    MADD rs, rt   MIPS32

Purpose:

    Multiply two words and add the result to Hi, Lo

Description:

    The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is added to the 64-bit concatenated values of *HI* and *LO* and the result is written back into *HI* and *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

    None.

Operation:

        temp <- (HI || LO) + (GPR[rs] * GPR[rt])

        HI <- $temp_{63..32}$

        LO <- $temp_{31..0}$

Exceptions:

    None

Programming Note:

    Where the size of the operands are known, software should place the shorter operand in GPR rt. The 4Kc and 4Km processor cores have a multiplier that has a lower latency when rt is a 16 bit value.

| Multiply and Add Unsigned Word to Hi,Lo | | | | | MADDU |

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 0 1 1 1 0 0	rs	rt	0 0 0 0 0 0	0 0 0 0 0 0	MADDU 0 0 0 0 0 1
6	5	5	5	5	6

Format:
    MADDU rs, rt   MIPS32

Purpose:
Multiply two unsigned words and add the result to Hi, Lo

Description:
The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is added to the 64-bit concatenated values of *HI* and *LO* and the result is written back into *HI* and *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:
None.

Operation:

$$\text{temp} \leftarrow (HI \ || \ LO) + ((0^{32} \ || \ GPR[rs]) * (0^{32} \ || \ GPR[rt]))$$

$$HI \leftarrow \text{temp}_{63..32}$$

$$LO \leftarrow \text{temp}_{31..0}$$

Exceptions:
None

Programming Note:
Where the size of the operands are known, software should place the shorter operand in GPR rt. The 4Kc and 4Km processor cores have a multiplier that has a lower latency when rt is a 16 bit value.

Move from Coprocessor 0   **MFC0**

31 26	25 21	20 16	15 11	10 3	2 0
COP0 0 1 0 0 0 0	MF 0 0 0 0 0	rt	rd	0 0 0 0 0 0 0 0 0	sel
6	5	5	5	8	3
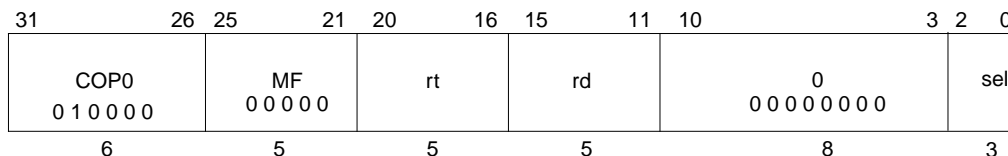
Format:

```
MFC0   rt, rd, sel                                              MIPS32
```

Purpose:

Move the contents of a coprocessor register to a general register.

Description:

The contents of the coprocessor 0 register specified by the combination of *rd* and *sel* are loaded into general register *rt*. Not all coprocessors or registers within a coprocessor support the sub-selection specified by the *sel* field. In those instances, the *sel* field must be set to zero

Restrictions:

For coprocessor 0, this instruction is legal only if the processor is in kernel mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

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

Operation:

if $(SR_{CUz} = 1)$ or

$((SR_{UM} = 0)$ or $(SR_{EXL} = 1)$ or $(SR_{ERL} = 1))$ then

```
                data <- CPR[z,rd,sel]
                GPR[rt] <- data
```

else

        InitiateCoprocessorUnusableException(0)

endif

Exceptions:

Coprocessor Unusable Exception

Move From HI Register MFHI

31 26	25 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0	rd	0 0 0 0 0 0	MFHI 0 1 0 0 0 0
6	10	5	5	6

Format: MFHI rd **MIPS I**

Purpose: To copy the special purpose *HI* register to a GPR

Description: rd ← HI

The contents of special register *HI* are loaded into GPR *rd*.

Restrictions: None

Operation:

GPR[rd] ← HI

Exceptions: None

Move From LO Register   **MFLO**

31 26	25 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0	rd	0 0 0 0 0 0	MFLO 0 1 0 0 1 0
6	10	5	5	6
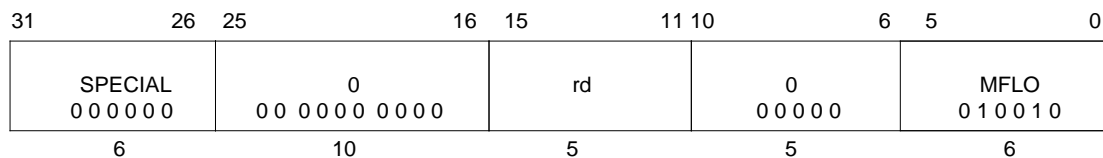
Format: MFLO   rd   **MIPS I**

Purpose: To copy the special purpose *LO* register to a GPR

Description: rd ← LO

The contents of special register *LO* are loaded into GPR *rd*.

Restrictions: None

Operation:

GPR[rd] ← LO

Exceptions: None

Move Conditional on Not Zero

MOVN

SPECIAL 000000	rs	rt	rd	0 00000	MOVN 001011
31 26	25 21	20 16	15 11	10 6	5 0
6	5	5	5	5	6

Format: MOVN   rd, rs, rt   **MIPS IV**

Purpose: To conditionally move a GPR after testing a GPR value

Description: if rt ≠ 0 then rd ← rs

If the value in GPR *rt* is not equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions: None

Operation:

```
if GPR[rt] ≠ 0 then
          GPR[rd] ← GPR[rs]
          endif
```

Exceptions: Reserved Instruction

Programming Notes:

The non-zero value tested here is the *condition true* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

Move Conditional on Zero   MOVZ

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL	rs	rt	rd	0	MOVZ
0 0 0 0 0 0				0 0 0 0 0	0 0 1 0 1 0
6	5	5	5	5	6

Format: MOVZ   rd, rs, rt   **MIPS IV**

Purpose: To conditionally move a GPR after testing a GPR value

Description: if rt = 0 then rd ← rs

If the value in GPR *rt* is equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions: None

Operation:

```
if GPR[rt] = 0 then
          GPR[rd] ← GPR[rs]
          endif
```

Exceptions: Reserved Instruction

Programming Notes:

The zero value tested here is the *condition false* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

Multiply and Subtract Word to Hi,Lo					**MSUB**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 0 1 1 1 0 0	rs	rt	0 0 0 0 0 0	0 0 0 0 0 0	MSUB 0 0 0 1 0 0
6	5	5	5	5	6
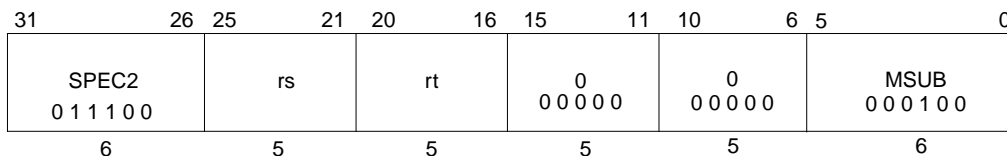
Format:

    MSUB   rs, rt   MIPS32

Purpose:

Multiply two words and subtract the result from Hi, Lo

Description:

The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values of *HI* and *LO* and the result is written back into *HI* and *LO*.   No arithmetic exception occurs under any circumstances.

Restrictions:

None.

Operation:

    temp <- (HI || LO) - (GPR[rs] * GPR[rt])

    HI <- temp$_{63..32}$

    LO <- temp$_{31..0}$

Exceptions:

None

Programming Note:

Where the size of the operands are known, software should place the shorter operand in GPR rt. The 4Kc and 4Km processor cores have a multiplier that has a lower latency when rt is a 16 bit value.

Multiply and Subtract Unsigned Word to Hi,Lo   **MSUBU**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 0 1 1 1 0 0	rs	rt	0 0 0 0 0 0	0 0 0 0 0 0	MSUBU 0 0 0 1 0 1
6	5	5	5	5	6

Format:
    MSUBU rs, rt   MIPS32

Purpose:
    Multiply two unsigned words and subtract the result from Hi, Lo

Description:
    The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values of *HI* and *LO* and the result is written back into *HI* and *LO*. No arithmetic exception occurs under any circumstances.

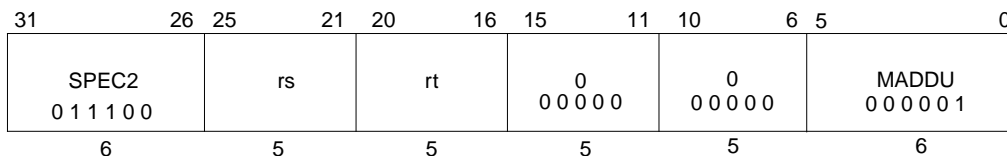**Restrictions:**
    None.

Operation:

$$\text{temp} \leftarrow (HI \mathbin{||} LO) - ((0^{32} \mathbin{||} GPR[rs]) * (0^{32} \mathbin{||} GPR[rt]))$$

$$HI \leftarrow \text{temp}_{63..32}$$

$$LO \leftarrow \text{temp}_{31..0}$$

Exceptions:
    None

Programming Note:
    Where the size of the operands are known, software should place the shorter operand in GPR rt. The 4Kc and 4Km processor cores have a multiplier that has a lower latency when rt is a 16 bit value.

Move to Coprocessor 0   **MTC0**

31 26	25 21	20 16	15 11	10 3	2 0
COP0 0 1 0 0 0 0	MT 0 0 1 0 0	rt	rd	0 0 0 0 0 0 0 0	sel
6	5	5	5	8	3
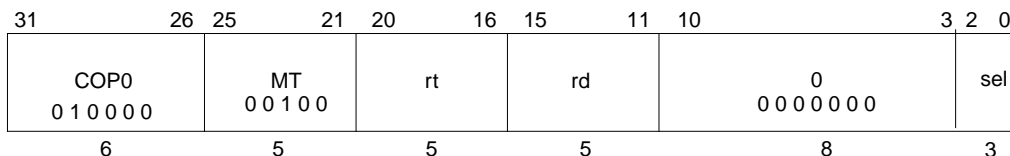
Format:
```
MTC0   rt, rd, sel                                                    MIPS32
```

Purpose:
Move the contents of a general register to a coprocessor register.

Description:
The contents of general register *rt* are loaded into the coprocessor z register specified by the combination of *rd* and *sel*. Not all coprocessors or registers within a coprocessor support the sub-selection specified by the *sel* field. In those instances, the *sel* field must be set to zero.

Restrictions:
For coprocessor 0, this instruction is legal only if the processor is in kernel mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

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

Operation:
if ($SR_{CUz} = 1$) or
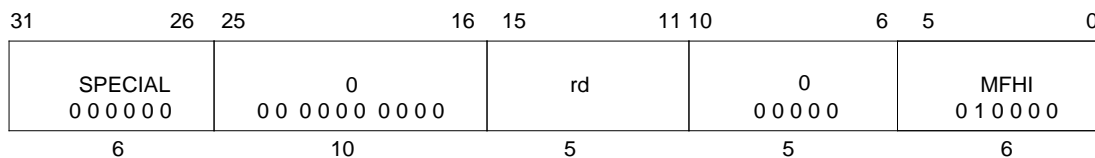(($SR_{UM} = 0$) or ($SR_{EXL} = 1$) or ($SR_{ERL} = 1$)) then
```
                data <- GPR[rt]
                CPR[z,rd,sel] <- data
```
else
    InitiateCoprocessorUnusableException(0)
endif

Exceptions:
   Coprocessor Unusable Exception

Move to HI Register
<div style="text-align: right">**MTHI**</div>

31 26	25 21	20 6	5 0
SPECIAL 0 0 0 0 0 0	rs	0 0 0000 0000 0000 00	MTHI 0 1 0 0 0 1
6	5	15	6
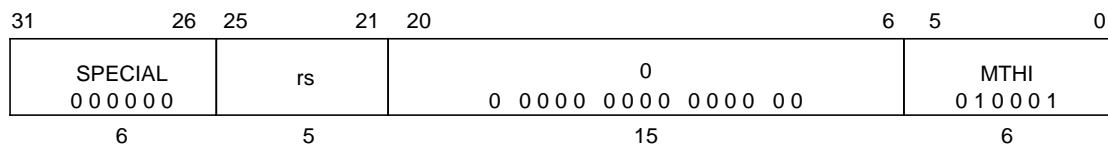
Format: MTHI   rs   **MIPS I**

Purpose: To copy a GPR to the special purpose *HI* register

Description: HI ← rs

The contents of GPR *rs* are loaded into special register *HI*.

Restrictions: None

Operation:

> HI ← GPR[rs]

Exceptions: None

Move to LO Register

MTLO

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 0 0000 0000 0000 00	MTLO 010011
6	5	15	6
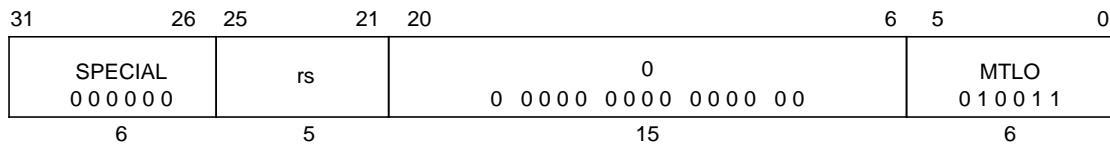
Format: MTLO   rs   **MIPS I**

Purpose: To copy a GPR to the special purpose *LO* register

Description: LO ← rs

The contents of GPR *rs* are loaded into special register *LO*.

Restrictions: None

Operation:

            LO ← GPR[rs]

Exceptions: None

Multiply Word to GPR					**MUL**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 011100	rs	rt	rd	0 00000	MUL 000010
6	5	5	5	5	6
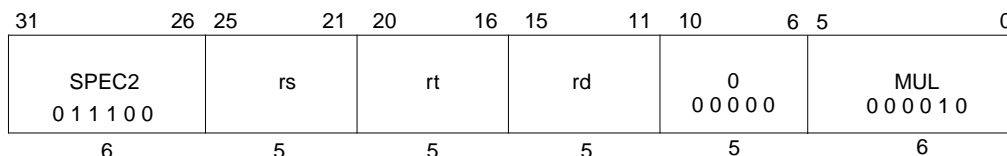
Format:
    MUL   rd, rs, rt   MIPS32

Purpose:
    Multiply two words write the result to a GPR

Description:
    The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The least significant 32 bits of the product are written to GPR *rd*. The contents of *HI* and *LO* are not defined after the operation. No arithmetic exception occurs under any circumstances.

Restrictions:
    Note that this instruction does not provide the capability of writing the result to the HI and LO registers. This is to prevent having two destination registers that would be difficult to support in potential high-performance processor implementations that rename registers.

Operation:
    temp <- GPR[rs] * GPR[rt]
    GPR[rd] <- $temp_{31..0}$
    HI <- **UNPREDICTABLE**
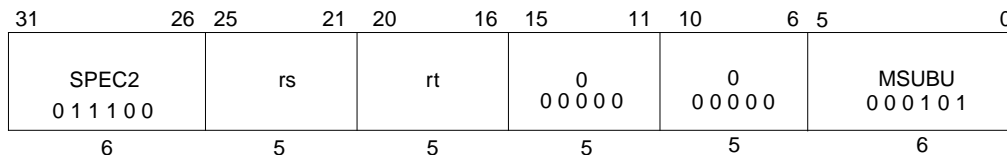   LO <- **UNPREDICTABLE**

Exceptions:
    None

Programming Note:
    Where the size of the operands are known, software should place the shorter operand in GPR rt. The 4Kc and 4Km processor cores have a multiplier that has a lower latency when rt is a 16 bit value.

Multiply Word

MULT

31	26	25	21	20	16	15	6	5	0
SPECIAL 0 0 0 0 0 0		rs		rt		0 0 0 0 0 0 0 0 0 0		MULT 0 1 1 0 0 0	
6		5		5		10		6	
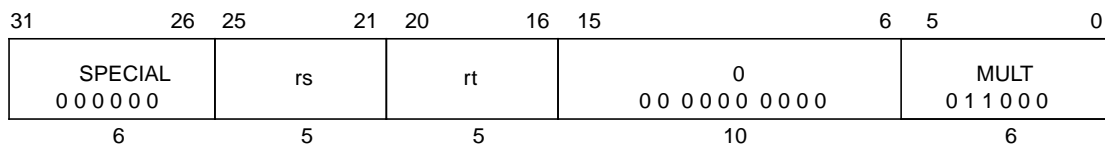
Format: MULT   rs, rt   **MIPS I**

Purpose: To multiply 32-bit signed integers

Description: (LO, HI) ← rs × rt

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register *LO*, and the high-order 32-bit word is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions: None

Operation:

```
prod← GPR[rs]₃₁..₀ × GPR[rt]₃₁..₀
LO← sign_extend(prod₃₁..₀)
HI← sign_extend(prod₆₃..₃₂)
```

Exceptions: None

Programming Notes:

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

Programs that require overflow detection must check for it explicitly.

Where the size of the operands are known, software should place the shorter operand in GPR rt. The 4Kc and 4Km processor cores have a multiplier that has a lower latency when rt is a 16 bit value.

Multiply Unsigned Word   MULTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	MULTU 011001
6	5	5	10	6
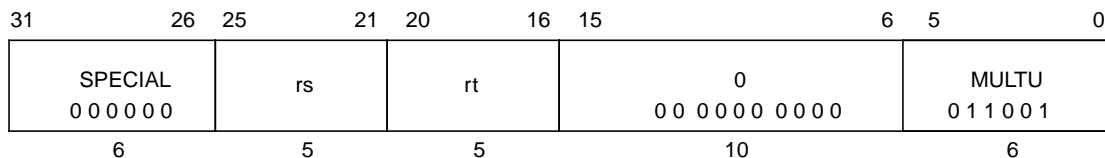
Format: MULTU   rs, rt   **MIPS I**

Purpose: To multiply 32-bit unsigned integers

Description: (LO, HI) ← rs × rt

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register *LO*, and the high-order 32-bit word is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions: None

Operation:

```
prod← (0 || GPR[rs]31..0) × (0 || GPR[rt]31..0)
LO← sign_extend(prod31..0)
HI← sign_extend(prod63..32
```

Exceptions: None

Programming Notes:

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

Programs that require overflow detection must check for it explicitly.

Where the size of the operands are known, software should place the shorter operand in GPR rt. The 4Kc and 4Km processor cores have a multiplier that has a lower latency when rt is a 16 bit value.

Not Or   **NOR**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	NOR 1 0 0 1 1 1
6	5	5	5	5	6
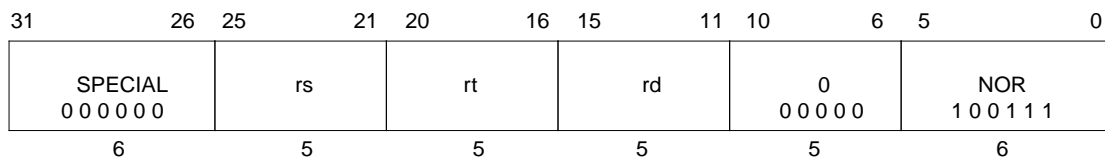
Format:　　　　　　NOR　　　　rd, rs, rt　　　　　　　　　　　　　　　　　　**MIPS I**

Purpose: To do a bitwise logical NOT OR

Description: `rd ← rs NOR rt`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical NOR operation. The result is placed into GPR *rd*.

Restrictions: None

Operation:

`GPR[rd] ← GPR[rs] nor GPR[rt]`

Exceptions: None

Or **OR**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	OR 1 0 0 1 0 1
6	5	5	5	5	6
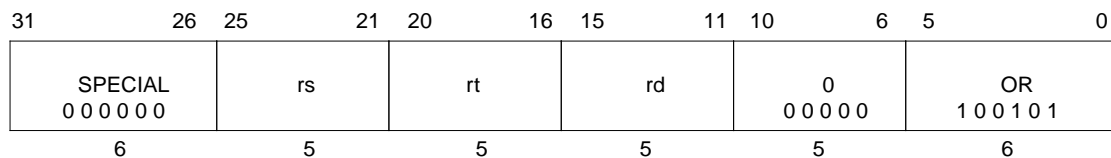
Format: OR   rd, rs, rt   **MIPS I**

Purpose: To do a bitwise logical OR

Description: rd ← rs or rt

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

Restrictions: None

Operation:

GPR[rd] ← GPR[rs] or GPR[rt]

Exceptions: None

Or Immediate

ORI

31	26	25	21	20	16	15	0
ORI 001101		rs		rt		immediate	
6		5		5		16	
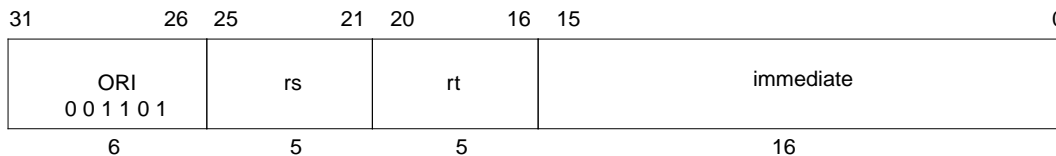
Format: ORI   rt, rs, immediate   **MIPS I**

Purpose: To do a bitwise logical OR with a constant

Description: `rd ← rs or immediate`

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

Restrictions: None

Operation:

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

Exceptions: None

Prefetch   **PREF**

31 PREF 26	25 21	20 16	15 0
110011	base	hint	offset
6	5	5	16
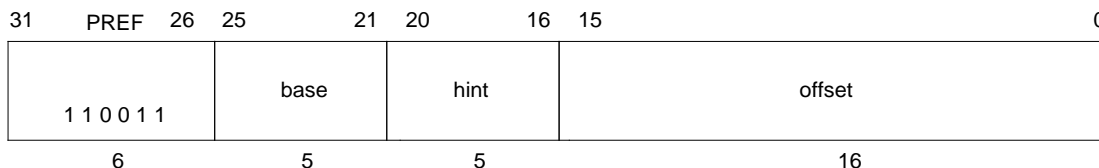
Format: PREF   hint, offset(base)   **MIPS IV**

Purpose: To prefetch data from memory

Description: `prefetch_memory(base+offset)`

PREF adds the 16-bit signed *offset* to the contents of GPR *base* to form an effective byte address. It advises that data at the effective address may be used in the near future. The *hint* field supplies information about the way that the data is expected to be used.

PREF is an advisory instruction that may change the performance of the program. However, for all *hint* values and all effective addresses, it neither changes the architecturally visible state nor does it alter the meaning of the program.

PREF does not cause addressing-related exceptions. If the address specified would cause an addressing exception, the exception condition is ignored and no data prefetch occurs.

PREF never generates a memory operation for a location with an *uncached* memory access type.

If PREF results in a memory operation, the memory access type used for the operation is determined by the memory access type of the effective address, just as it would be if the memory operation had been caused by a load or store to the effective address.

The *hint* field supplies information about the way the data is expected to be used. A *hint* value cannot cause an action to modify architecturally visible state. A processor may use a *hint* value to improve the effectiveness of the prefetch action.

Prefetch (cont.)

<div align="right">

PREF

</div>

Any of the following conditions causes the core to treat a PREF instruction as a NOP.

–   A reserved hint value is used

–   Writeback-invalidate (25) hint value is used

–   The address has a translation error

–   The address maps to an uncacheable page

–   The data is already in the cache

–   There is already another load/prefetch outstanding

In all other cases execution of the PREF instruction initiates an external bus read transaction. PREF is a non-blocking operation and does not cause the pipeline to stall while waiting for the data to be returned.

Table 11-11 Values of Hint Fields for the PREF Instruction

Value	Name	Data use and desired prefetch action
0	load	Data is expected to be loaded (not modified).
1	store	Data is expected to be stored or modified.
2-3		Reserved. Treated as a NOP
4	load_streamed	Data is expected to be loaded (not modified) but not reused extensively; it "streams" through cache.
5	store_streamed	Data is expected to be stored or modified but not reused extensively; it "streams" through cache. Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as "retained."
6	load_retained	Data is expected to be loaded (not modified) and reused extensively; it should be "retained" in the cache. Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as "streamed."
7	store_retained	Data is expected to be stored or modified and reused extensively; it should be "retained" in the cache.
8-24		Reserved. Treated as a NOP
25	writeback_invalidate	MIPS32 4K processor cores treat this hint as a NOP.
26-31		Reserved. Treated as a NOP
Reserved hint values and writeback_invalidate are treated as NOPs. All other hint values are treated the same - filling the cache if the conditions listed on the previous page are met.		

Restrictions: None

Prefetch (cont.)	PREF

Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
            (pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
            Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions: None

Programming Notes:

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

Prefetch does not cause addressing exceptions. It does not cause an exception to prefetch using an address pointer value before the validity of a pointer is determined.

Store Byte	SB

31	26	25	21	20	16	15	0
SB 101000		base		rt		offset	
6		5		5		16	
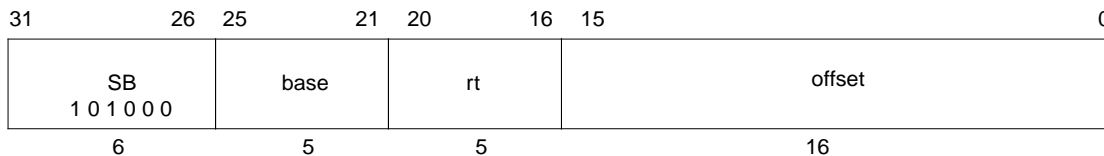
Format: SB   rt, offset(base)   **MIPS I**

Purpose: To store a byte to memory

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

The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions: None

Operation:

```
vAddr                                          ← sign_extend(offset) + GPR[base]
        (pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
        pAddr← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
        byte← vAddr_1..0 xor BigEndianCPU^2
        dataword← GPR[rt]_31-8*byte..0 || 0^8*byte
        StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
```

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

Store Conditional Word

31	26	25	21	20	16	15	0
SC 1 1 1 0 0 0		base		rt		offset	
6		5		5		16	
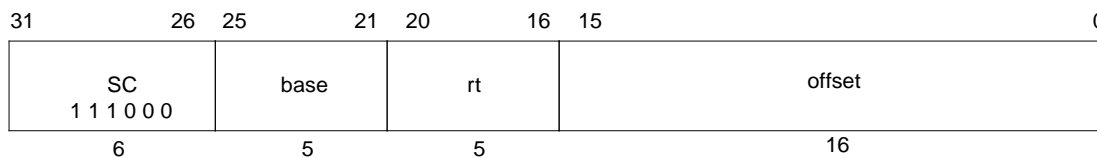
Format: SC rt, offset(base) **MIPS II**

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

Description: `if atomic_update then memory[base+offset]` ← rt, rt ← 1
`else rt` ← 0

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

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The SC completes the RMW sequence begun by the preceding LL instruction executed on the processor. To complete the RMW sequence atomically, the following occur:

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

- A 1, indicating success, is written into GPR *rt*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *rt*.

If the following event occurs between the execution of LL and SC, the SC fails and an exception occurs on the processor as detected by execution of the ERET instruction.

Store Conditional Word (cont.)	SC

The following conditions must be true or the result of the SC is undefined:

- Execution of SC must have been preceded by execution of an LL instruction.

- A RMW sequence executed without intervening exceptions must use the same address in the LL and SC. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.

Atomic RMW is provided only for cached memory locations. The extent to which the detection of atomicity operates correctly depends on the system implementation and the memory access type used for the location:

- Uniprocessor atomicity**:** To provide atomic RMW on a single processor, all accesses to the location must be made with memory access type of either *cached noncoherent* or *cached coherent*. All accesses must be to one or the other access type, and they may not be mixed.

Restrictions:

The addressed location must have a memory access type of *cached noncoherent* or *cached coherent*; if it does not, the result is undefined.

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

Operation:

```
vAddr                        ← sign_extend(offset) + GPR[base]
          if vAddr₁..₀ ≠ 0² then SignalException(AddressError) endif
          (pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
          dataword← GPR[rt]
          if LLbit then
          StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
          endif
          GPR[rt]← 0³¹ || LLbit
```

Exceptions: TLB Refill, TLB Invalid, TLB Modified, Address Error, Reserved Instruction

Store Conditional Word (cont.)	SC

Programming Notes:

LL and SC are used to atomically update memory locations, as shown in Figure 11-8.

```
L1:
LLT1, (T0)# load counter
ADDIT2, T1, 1# increment
SCT2, (T0)# try to store, checking for atomicity
BEQT2, 0, L1# if not atomic (0), try again
NOP# branch-delay slot
```

Figure 11-8 Example of LL/SC Atomic Update

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

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

Software Debug Breakpoint   **SDBBP**

31 26	25 6	5 0
SPEC2 0 1 1 1 0 0	code	SDBBP 1 1 1 1 1 1
6	20	6
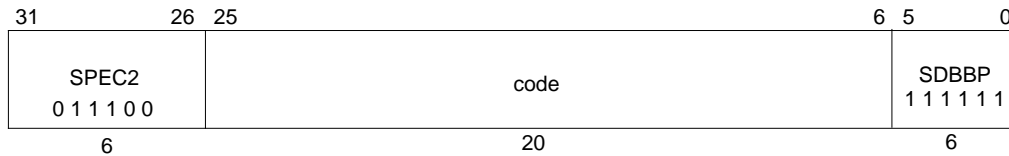
Format:

```
SDBBP code
```

Purpose:

To cause a debug software breakpoint exception.

Description:

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

The code field is available as software parameter, but is retrieved by the debug exception handler only by loading the contents of the memory containing the instruction.

Restrictions:

The operation of the processor is **UNDEFINED** if a SDBBP is executed in debug mode.

Operation:

```
SignalException(DebugSoftwareBreakpoint)
```

Exceptions:

Debug Software Breakpoint Exception

Store Halfword

SH

31	26	25	21	20	16	15	0
SH 1 0 1 0 0 1		base		rt		offset	
6		5		5		16	
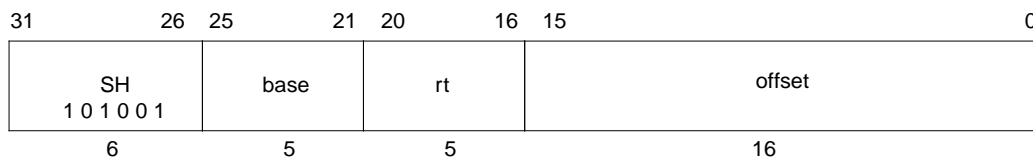
Format: SH rt, offset(base) **MIPS I**

Purpose: To store a halfword to memory

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

The least-significant 16-bit halfword of register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

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

Operation:

```
vAddr                     ← sign_extend(offset) + GPR[base]
         if vAddr₀ ≠ 0 then SignalException(AddressError) endif
         (pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
         pAddr← pAddr_PSIZE-1..2 || (pAddr_1..0 xor (ReverseEndian || 0))
         byte← vAddr_1..0  xor (BigEndianCPU || 0)
         dataword← GPR[rt]_31-8*byte..0 || 0^8*byte
         StoreMemory (CCA, HALFWORD, dataword, pAddr, vAddr, DATA)
```

Exceptions: TLB Refill, TLB Invalid, TLB Modified, Address Error

| Shift Word Left Logical | SLL |

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000		0 00000		rt		rd		sa		SLL 000000	
6		5		5		5		5		6	
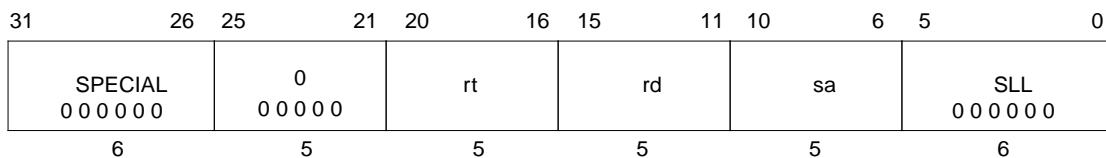
Format: SLL   rd, rt, sa   **MIPS I**

Purpose: To left-shift a word by a fixed number of bits

Description: rd ← rt << sa

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa.*

Restrictions: None

Operation:

```
s                          ← sa
        temp← GPR[rt](31-s)..0 || 0^s
        GPR[rd]← sign_extend(temp)
```

Exceptions: None

Programming Notes:

Some assemblers, particularly 32-bit assemblers, treat an SLL with a shift amount of zero as a NOP and either delete it or replace it with an actual NOP.

Shift Word Left Logical Variable <div align="right">**SLLV**</div>

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SLLV 0 0 0 1 0 0
6	5	5	5	5	6
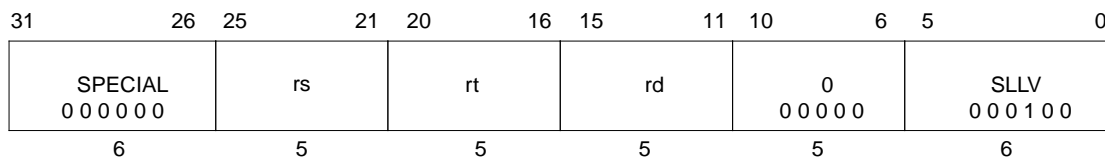
Format: SLLV   rd, rt, rs   **MIPS I**

Purpose: To left-shift a word by a variable number of bits

Description: rd ← rt << rs

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result word is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions: None

Operation:

$$s \leftarrow GPR[rs]_{4..0}$$
$$temp \leftarrow GPR[rt]_{(31-s)..0} \mathbin{\|} 0^{s}$$
$$GPR[rd] \leftarrow sign\_extend(temp)$$

Exceptions: None

Programming Notes: None

None

| **Set on Less Than** | | | | | | **SLT** |

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 0 0 0 0 0 0		rs		rt		rd		0 0 0 0 0 0		SLT 1 0 1 0 1 0	
6		5		5		5		5		6	
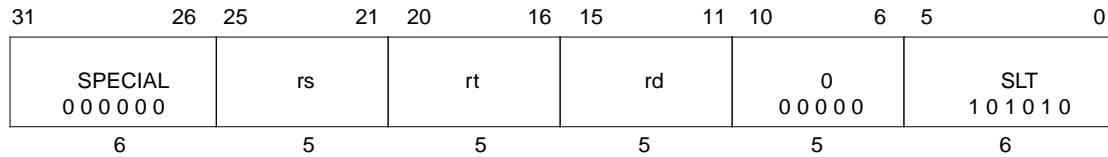
Format: SLT rd, rs, rt **MIPS I**

Purpose: To record the result of a less-than comparison

Description: rd ← (rs < rt)

Compare the contents of GPR *rs* and GPR *rt* as signed integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt,* the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions: None

Operation:

```
if GPR[rs] < GPR[rt] then
          GPR[rd] ← 0^GPRLEN-1 || 1
          else
          GPR[rd] ← 0^GPRLEN
          endif
```

Exceptions: None

Set on Less Than Immediate **SLTI**

31 26	25 21	20 16	15 0
SLTI 001010	rs	rt	immediate
6	5	5	16

Format: SLTI rt, rs, immediate **MIPS I**

Purpose: To record the result of a less-than comparison with a constant

Description: rt ← (rs < immediate)

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate,* the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions: None

Operation:

```
if GPR[rs] < sign_extend(immediate) then
        GPR[rd] ← 0^GPRLEN-1|| 1
        else
        GPR[rd] ← 0^GPRLEN
        endif
```

Exceptions: None

Set on Less Than Immediate	SLTI

31 26	25 21	20 16	15 0
SLTI 0 0 1 0 1 0	rs	rt	immediate
6	5	5	16

Format: SLTI   rt, rs, immediate   **MIPS I**

Purpose: To record the result of a less-than comparison with a constant

Description: rt ← (rs < immediate)

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate,* the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions: None

Operation:

```
if GPR[rs] < sign_extend(immediate) then
          GPR[rd] ← 0^(GPRLEN-1)|| 1
          else
          GPR[rd] ← 0^(GPRLEN)
          endif
```

Exceptions: None

SLTIU 0 0 1 0 1 1	rs	rt	immediate
6	5	5	16

31   26   25   21   20   16   15   0

Set on Less Than Immediate Unsigned   **SLTIU**

Format: SLTIU   rt, rs, immediate   **MIPS I**

Purpose: To record the result of an unsigned less-than comparison with a constant

Description: rt ← (rs < immediate)

Compare the contents of GPR *rs* and the sign-extended 16-bit *immediate* as unsigned integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate,* the result is 1 (true); otherwise, it is 0 (false).

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions: None

Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
        GPR[rd] ← 0^GPRLEN-1 || 1
        else
        GPR[rd] ← 0^GPRLEN
        endif
```

Exceptions: None

Set on Less Than Immediate Unsigned				SLTIU

31 26	25 21	20 16	15 0
SLTIU 0 0 1 0 1 1	rs	rt	immediate
6	5	5	16

Format: SLTIU   rt, rs, immediate   **MIPS I**

Purpose: To record the result of an unsigned less-than comparison with a constant

Description: rt ← (rs < immediate)

Compare the contents of GPR *rs* and the sign-extended 16-bit *immediate* as unsigned integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate,* the result is 1 (true); otherwise, it is 0 (false).

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions: None

Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
        GPR[rd] ← 0^{GPRLEN-1} || 1
        else
        GPR[rd] ← 0^{GPRLEN}
        endif
```

Exceptions: None

Set on Less Than Unsigned   SLTU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SLTU 1 0 1 0 1 1
6	5	5	5	5	6
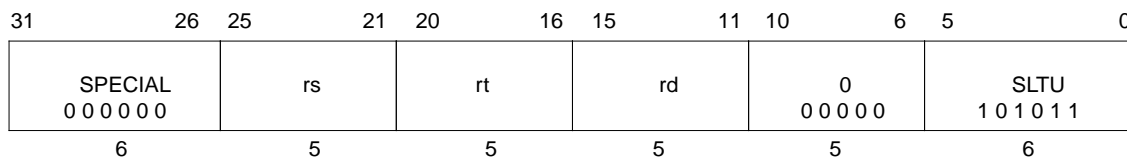
Format: SLTU   rd, rs, rt   **MIPS I**

Purpose: To record the result of an unsigned less-than comparison

Description: rd ← (rs < rt)

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt,* the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.
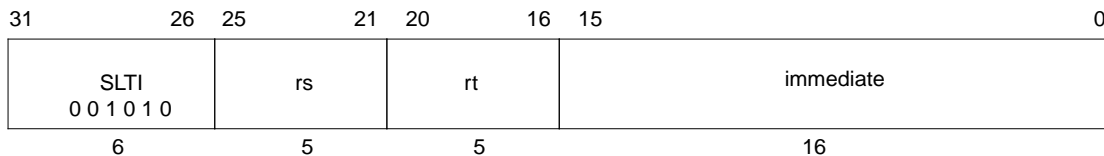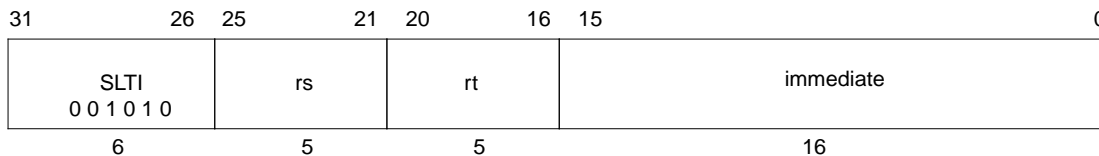
Restrictions: None

Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
        GPR[rd] ← 0GPRLEN-1 || 1
        else
        GPR[rd] ← 0GPRLEN
        endif
```

Exceptions: None

Shift Word Right Arithmetic SRA

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRA 000011
6	5	5	5	5	6
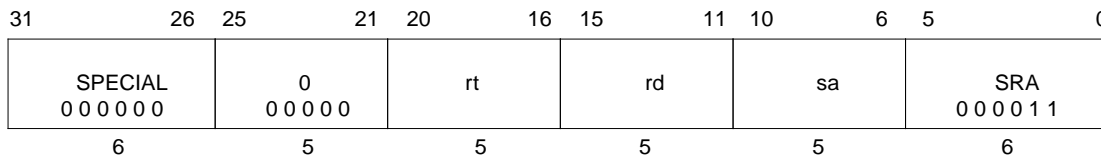
Format: SRA   rd, rt, sa   **MIPS I**

Purpose: To execute an arithmetic right-shift of a word by a fixed number of bits

Description: rd ← rt >> sa   (arithmetic)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa.*

Restrictions:

None

Operation:

```
s← sa
temp← (GPR[rt]₃₁)ˢ || GPR[rt]₃₁..s
GPR[rd]← sign_extend(temp)
```

Exceptions: None

Shift Word Right Arithmetic Variable | SRAV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SRAV 000111
6	5	5	5	5	6
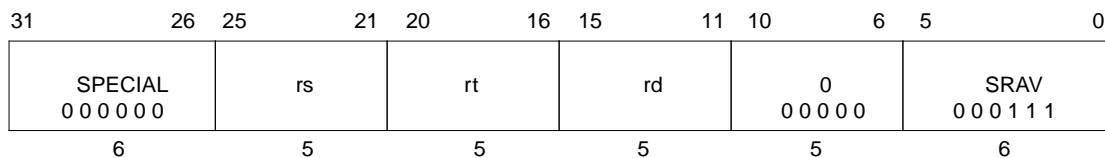
Format: SRAV   rd, rt, rs   **MIPS I**

Purpose: To execute an arithmetic right-shift of a word by a variable number of bits

Description: rd ← rt >> rs   (arithmetic)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

None

Operation:

$$s \leftarrow GPR[rs]_{4..0}$$
$$temp \leftarrow (GPR[rt]_{31})^s \ || \ GPR[rt]_{31..s}$$
$$GPR[rd] \leftarrow sign\_extend(temp)$$

Exceptions: None

Shift Word Right Logical **SRL**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRL 000010
6	5	5	5	5	6
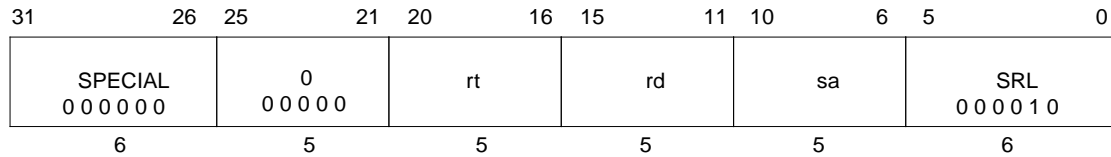
Format: SRL   rd, rt, sa   **MIPS I**

Purpose: To execute a logical right-shift of a word by a fixed number of bits

Description: rd ← rt >> sa   (logical)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions: None

Operation:

```
s← sa
temp← 0ˢ || GPR[rt]₃₁..ₛ
GPR[rd]← sign_extend(temp)
```

Exceptions: None

Shift Word Right Logical Variable					**SRLV**

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 0 0 0 0 0 0		rs		rt		rd		0 0 0 0 0 0		SRLV 0 0 0 1 1 0	
6		5		5		5		5		6	
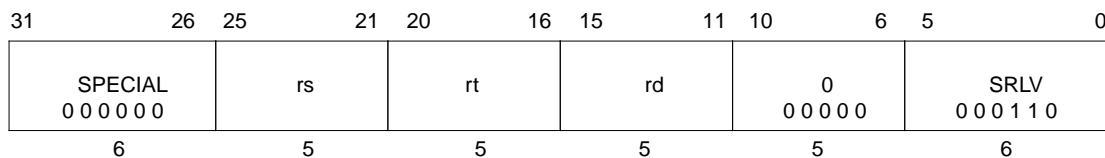
Format: SRLV rd, rt, rs **MIPS I**

Purpose: To execute a logical right-shift of a word by a variable number of bits

Description: rd ← rt >> rs (logical)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

None

Operation:

$$s \leftarrow GPR[rs]_{4..0}$$
$$temp \leftarrow 0^s \,||\, GPR[rt]_{31..s}$$
$$GPR[rd] \leftarrow sign\_extend(temp)$$

Exceptions: None

Subtract Word   SUB

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUB 100010
6	5	5	5	5	6
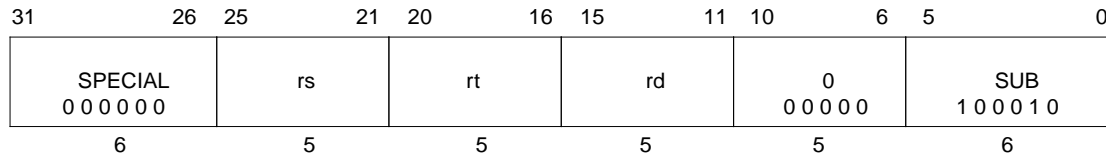
Format: SUB   rd, rs, rt   **MIPS I**

Purpose: To subtract 32-bit integers. If overflow occurs, then trap

Description: rd ← rs - rt

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* to produce a 32-bit result. If the subtraction results in 32-bit 2's complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions: None

Operation:

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

Exceptions: Integer Overflow

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

Subtract Unsigned Word | SUBU

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000		rs		rt		rd		0 00000		SUBU 100011	
6		5		5		5		5		6	
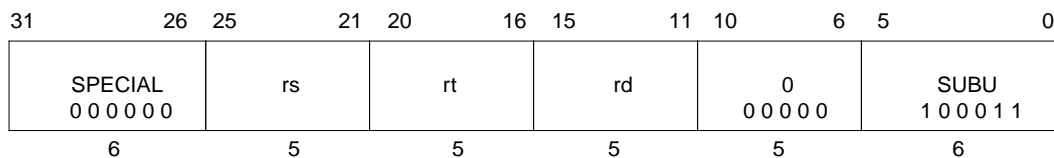
Format: SUBU rd, rs, rt   **MIPS I**

Purpose: To subtract 32-bit integers

Description: rd ← rs - rt

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

No integer overflow exception occurs under any circumstances.

Restrictions: None

Operation:

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

Exceptions: None

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

Store Word

<div align="right">SW</div>

31	26	25	21	20	16	15	0
SW 1 0 1 0 1 1		base		rt		offset	
6		5		5		16	
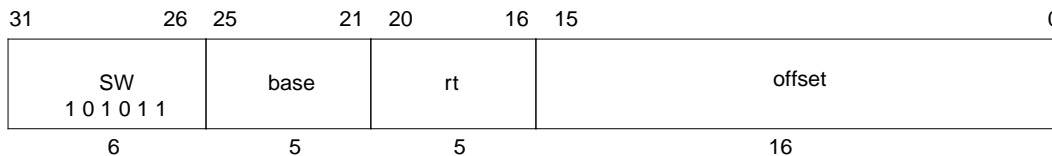
Format: SW rt, offset(base) **MIPS I**

Purpose: To store a word to memory

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

The least-significant 32-bit word of register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

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

Operation:

```
vAddr                        ← sign_extend(offset) + GPR[base]
            if vAddr_{1..0} ≠ 0^2 then SignalException(AddressError) endif
            (pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
            dataword← GPR[rt]
            StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions: TLB Refill, TLB Invalid, TLB Modified, Address Error

Store Word Left · SWL

31 26	25 21	20 16	15 0
SWL 1 0 1 0 1 0	base	rt	offset
6	5	5	16
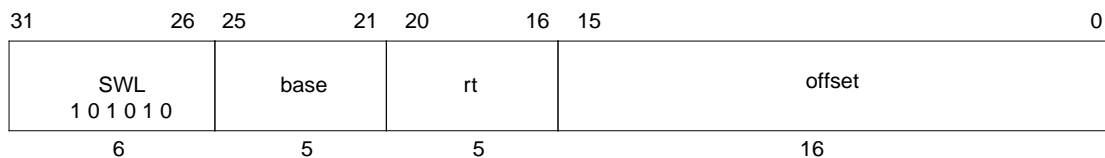
Format: SWL   rt, offset(base)   **MIPS I**

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

Description: memory[base+offset] ← rt

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr)*. *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W)* in memory starting at an arbitrary byte boundary.

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

Figure 11-9 illustrates this operation using big-endian byte ordering for 32-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWL stores the most-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWR stores the remainder of the unaligned word.

Store Word Left (cont.)	SWL

Figure 11-9 Unaligned Word Store Using SWL and SWR

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address (*vAddr1..0*)—and the current byte-ordering mode of the processor (big- or little-endian). Figure 11-10 shows the bytes stored for every combination of offset and byte ordering.

Store Word Left (cont.)

Memory contents and byte offsets   Initial contents of Dest Register

0   1   2   3   ←big-endian

| i | j | k | l |

offset ($vAddr_{1..0}$)

3   2   1   0   ←little-endian   most   — significance — least

most   least   32-bit register | E | F | G | H |

— significance —

Memory contents after instruction (shaded is unchanged)

Big-endian $vAddr_{1..0}$ Little-endian
byte ordering   byte ordering

E	F	G	H	0
i	E	F	G	1
i	j	E	F	2
i	j	k	E	3

i	j	k	E
i	j	E	F
i	E	F	G
E	F	G	H

Figure 11-10 Bytes Stored by an SWL Instruction

Restrictions: None

Store Word Left (cont.)	SWL

Operation:

```
vAddr                    ← sign_extend(offset) + GPR[base]
          (pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
          pAddr← pAddr_PSIZE-1..2 || (pAddr_1..0  xor  ReverseEndian²)
          If BigEndianMem = 0 then
          pAddr ← pAddr_PSIZE-1..2 || 0²
          endif
          byte← vAddr_1..0 xor BigEndianCPU²
          dataword← 0^(24-8*byte) || GPR[rt]_31..24-8*byte
          StoreMemory (CCA, byte, dataword, pAddr, vAddr, DATA)
```

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

Store Word Right

SWR

31 26	25 21	20 16	15 0
SWR 1 0 1 1 1 0	base	rt	offset
6	5	5	16
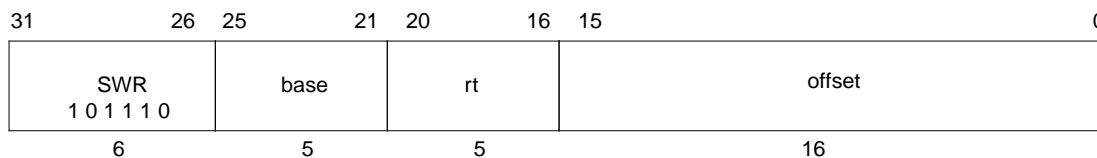
Format: SWR   rt, offset(base)   **MIPS I**

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

Description: memory[base+offset] ← rt

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr)*. *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word *(W)* in memory starting at an arbitrary byte boundary.

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

Figure 11-11 illustrates this operation using big-endian byte ordering for 32-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR stores the least-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWL stores the remainder of the unaligned word.

Word at byte 2 in memory, big-endian byte order, each mem byte contains its address

most   — significance —   least

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... | Memory: Initial contents

E   F   G   H

32-bit GPR 24 | E | F | G | H |

| 0 | 1 | 2 | 3 | G | H | 6 | ... | After executing SWR $24,5($0)

| 0 | 1 | E | F | G | H | 6 | ... | Then after SWL $24,2($0)

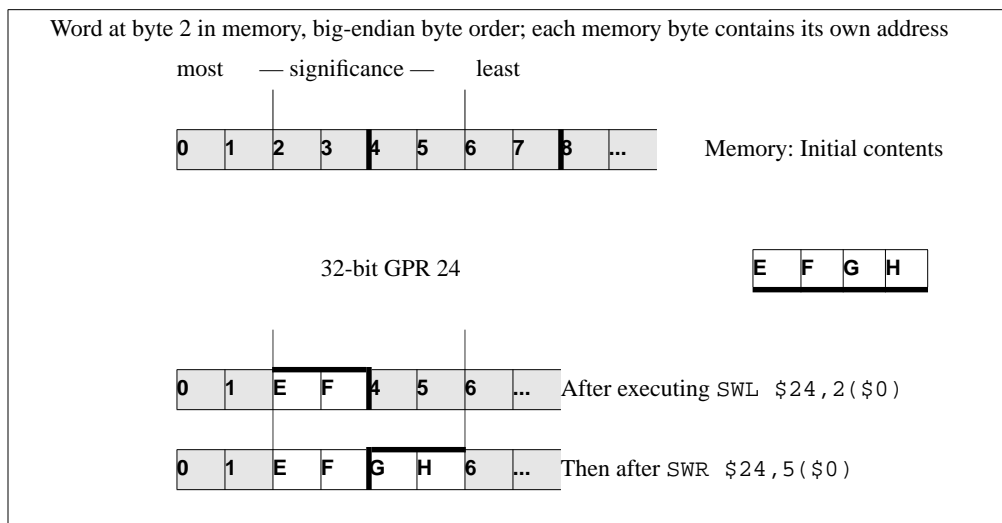**Figure 11-11 Unaligned Word Store Using SWR and SWL**

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address (*vAddr1..0*)—and the current byte-ordering mode of the processor (big- or little-endian). Figure 11-12 shows the bytes stored for every combination of offset and byte-ordering.
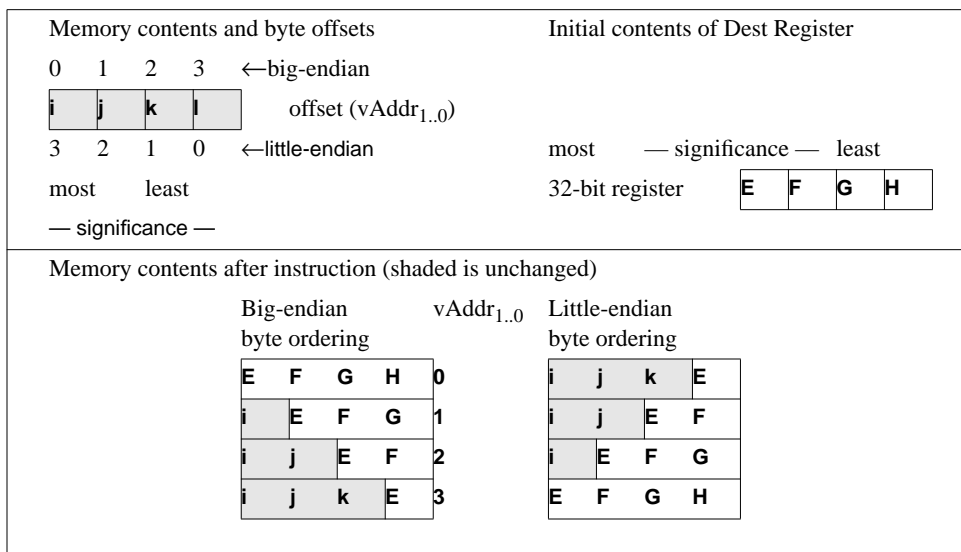
Store Word Right (cont.)	SWR

Memory contents and byte offsets   Initial contents of Dest Register

0   1   2   3   ← big-endian

| i | j | k | l | offset ($vAddr_{1..0}$)

3   2   1   0   ← little-endian   most   — significance —   least

most   least   32-bit register | E | F | G | H |

— significance —

Memory contents after instruction (shaded is unchanged)

Big-endian $vAddr_{1..0}$ Little-endian byte
byte ordering   ordering

H	j	k	l	0		E	F	G	H
G	H	k	l	1		F	G	H	I
F	G	H	l	2		G	H	k	I
E	F	G	H	3		H	j	k	l
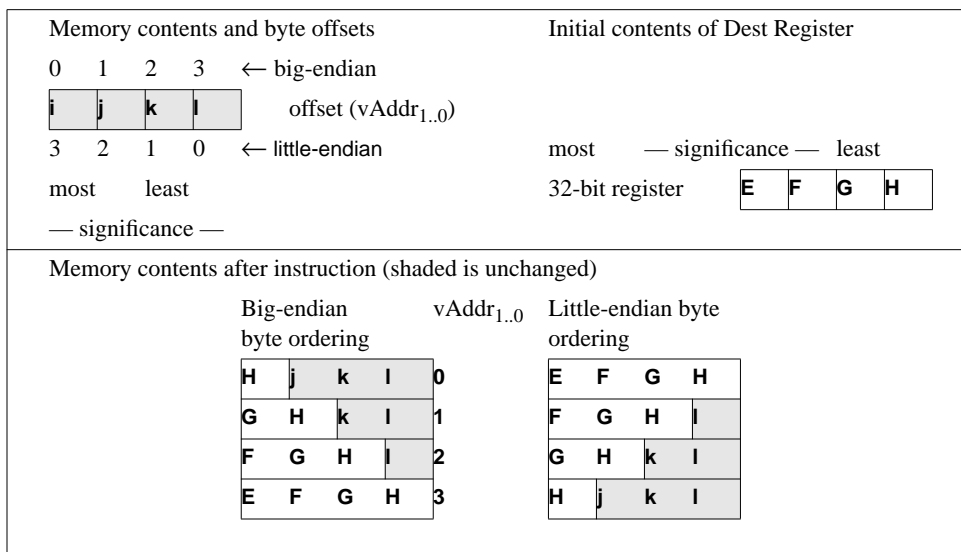
Figure 11-12 Bytes Stored by SWR Instruction

**Restrictions:** None

Operation:

```
vAddr                    ← sign_extend(offset) + GPR[base]
        (pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
        pAddr← pAddr_PSIZE-1..2 || (pAddr_1..0  xor  ReverseEndian²)
        if BigEndianMem = 0 then
        pAddr← pAddr_PSIZE-1..2 || 0²
        endif
        byte← vAddr_1..0  xor  BigEndianCPU²
        dataword← GPR[rt]_31-8*byte || 0^8*byte
        StoreMemory (CCA, WORD-byte, dataword, pAddr, vAddr, DATA)
```

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

Synchronize Shared Memory   SYNC

31 26	25 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 00 0000 0000 0000 0	stype	SYNC 0 0 1 1 1 1
6	15	5	6
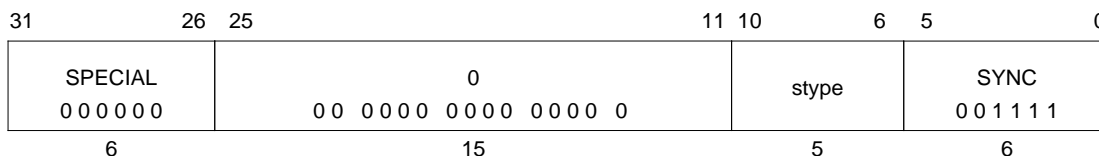
Format: SYNC   (stype = 0 implied)   **MIPS II**

Purpose: To order loads and stores.

Description:

The SYNC instruction affects only *uncached* and *cached coherent* loads and stores. The loads and stores that occur before the SYNC must be completed before the loads and stores after the SYNC are allowed to start. Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.

SYNC does not guarantee the order in which instruction fetches are performed. The *stype* values 1-31 are reserved; they produce the same result as the value zero. Executing a SYNC instruction causes the write-through buffer to be flushed. The SYNC instruction stalls until all loads and stores are completed.

Restrictions:

The effect of SYNC on the global order of loads and stores for memory access types other than *uncached* and *cached coherent* is not defined.

Operation:

SyncOperation(stype)

Exceptions: None

Programming Note:

The description above refers to the 4K core implementation of the SYNC instruction. For a more detailed description of the programming effects of SYNC on a generic MIPS32 processor, refer to the MIPS32 Specification.

System Call			SYSCALL

31	26	25	6	5	0
SPECIAL 0 0 0 0 0 0		code		SYSCALL 0 0 1 1 00	
6		20		6	
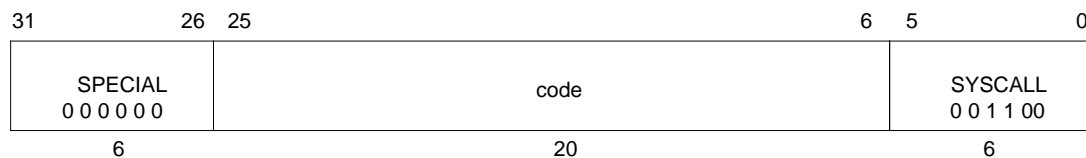
Format: SYSCALL   **MIPS I**

Purpose: To cause a System Call exception

Description:

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

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

Restrictions: None

Operation:

SignalException(SystemCall)

Exceptions: System Call

Trap if Equal   TEQ

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TEQ 1 1 0 1 0 0
6	5	5	10	6
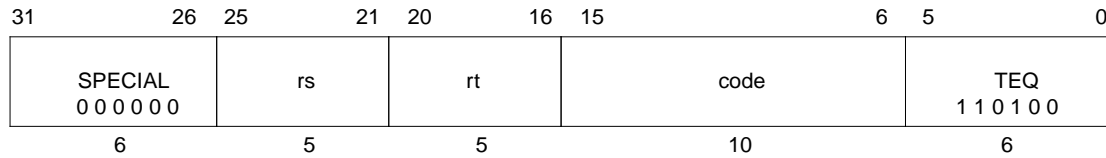
Format: TEQ   rs, rt   **MIPS II**

Purpose: To compare GPRs and do a conditional trap

Description: `if rs = rt then Trap`

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is equal to GPR *rt,* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions: None

Operation:

```
if GPR[rs] = GPR[rt] then
         SignalException(Trap)
         endif
```

Exceptions: Trap

Trap if Equal Immediate   **TEQI**

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TEQI 0 1 1 0 0	immediate
6	5	5	16
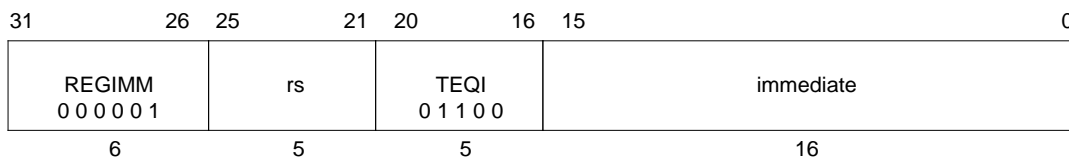
Format: TEQI   rs, immediate   **MIPS II**

Purpose: To compare a GPR to a constant and do a conditional trap

Description: if rs = immediate then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is equal to *immediate,* then take a Trap exception.

Restrictions: None

Operation:

```
if GPR[rs] = sign_extend(immediate) then
          SignalException(Trap)
          endif
```

Exceptions: Trap

Trap if Greater or Equal   TGE

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TGE 1 1 0 0 0 0
6	5	5	10	6

Format: TGE   rs, rt   **MIPS II**

Purpose: To compare GPRs and do a conditional trap

Description: if rs ≥ rt then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is greater than or equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions: None

Operation:

```
if GPR[rs] ≥ GPR[rt] then
          SignalException(Trap)
          endif
```

Exceptions: Trap

Trap if Greater or Equal Immediate

TGEI

REGIMM 000001	rs	TGEI 01000	immediate
6	5	5	16

31   26 25   21 20   16 15   0
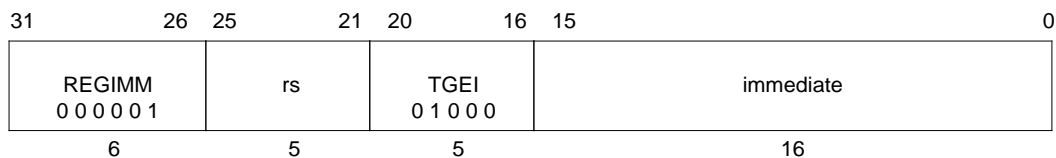
Format: TGEI   rs, immediate   **MIPS II**

Purpose: To compare a GPR to a constant and do a conditional trap

Description: if rs ≥ immediate then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is greater than or equal to *immediate*, then take a Trap exception.

Restrictions: None

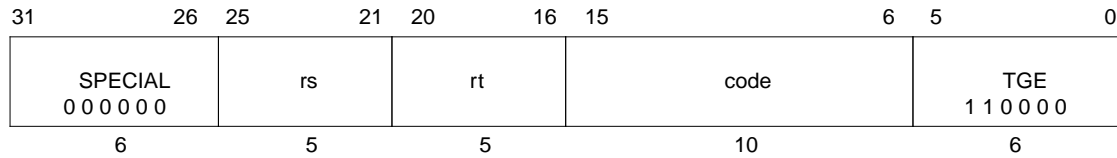**Operation:**

```
if GPR[rs] ≥ sign_extend(immediate) then
          SignalException(Trap)
          endif
```

Exceptions: Trap

Trap if Greater or Equal Immediate Unsigned				**TGEIU**

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TGEIU 0 1 0 0 1	immediate
6	5	5	16
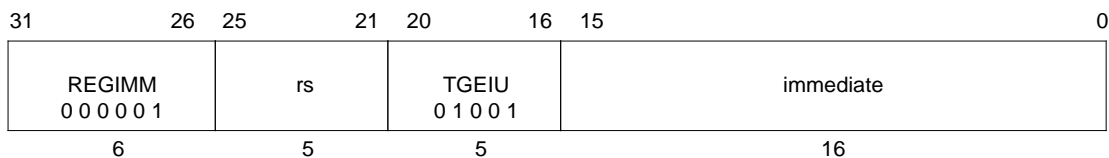
Format: TGEIU   rs, immediate   **MIPS II**

Purpose: To compare a GPR to a constant and do a conditional trap

Description: if rs ≥ immediate then Trap

Compare the contents of GPR *rs* and the 16-bit sign-extended *immediate* as unsigned integers; if GPR *rs* is greater than or equal to *immediate*, then take a Trap exception.

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

Restrictions: None

Operation:

```
if (0 || GPR[rs]) ≥ (0 || sign_extend(immediate)) then
          SignalException(Trap)
          endif
```

Exceptions: Trap

Trap if Greater or Equal Unsigned — TGEU

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TGEU 1 1 0 0 0 1
6	5	5	10	6
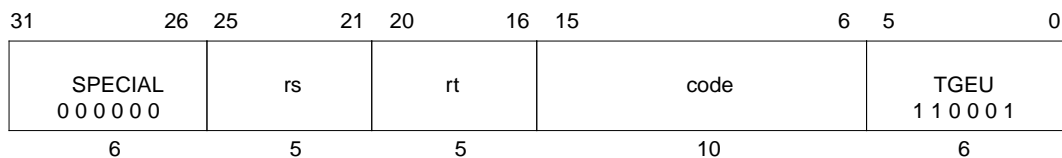
Format: TGEU rs, rt **MIPS II**

Purpose: To compare GPRs and do a conditional trap

Description: if rs ≥ rt then Trap

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; if GPR *rs* is greater than or equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions: None

Operation:

```
if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
        SignalException(Trap)
        endif
```

Exceptions: Trap

Probe TLB for Matching Entry				TLBP

31 26	25 24			0
COP0 010000	CO 1	0 000 0000 0000 0000 0000		TLBP 001000
6	1	19		6
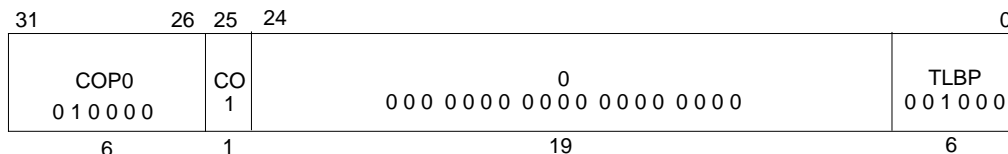
Format:
```
TLBP                                                            MIPS32
```

Purpose:
Find a matching entry in the TLB.

Description:
The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the *Index* register is set.

Restrictions:
This instruction is legal only if the processor is in kernel mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

For processors that do not include the standard TLB MMU, the operation of this instruction is **UNDEFINED**.

Operation:
```
if (SR_CU0 = 1) or (SR_UM = 0) or (SR_EXL = 1) or (SR_ERL = 1) then
    Index <- 1 || UNPREDICTABLE^31
    for i in 0...TLBEntries-1
        if((TLB[i]_VPN2 and not (TLB[i]_Mask)) =
            (EntryHi_VPN2 and not (TLB[i]_Mask))) and
            (TLB[i]_G or (TLB[i]_ASID = EntryHi_ASID)) then
                Index <- i
        endif
    endfor
else
    InitiateCoprocessorUnusableException(0)
endif
```

Exceptions:
Coprocessor Unusable Exception

Read Indexed TLB Entry   TLBR

31	26	25	24	0

COP0 0 1 0 0 0 0	CO 1	0 000 0000 0000 0000 0000	TLBR 000001
6	1	19	6
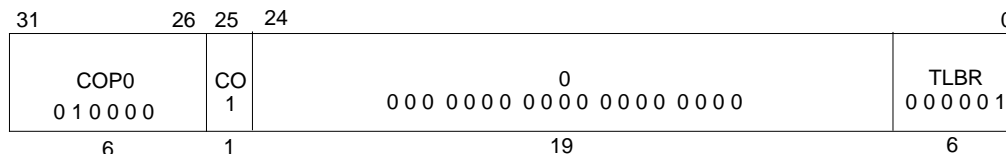
Format:

TLBR   MIPS32

Purpose:

Read an entry from the TLB.

Description:

The *EntryHi*, *EntryLo0*, *EntryLo1,* and *PageMask* registers are loaded with the contents of the TLB entry pointed to by the *Index* register. Note that the value written to the EntryHi, EntryLo0, and EntryLo1 registers may be different from that originally written to the TLB via these registers in that:

- **The value returned in the VPN2 field of the *EntryHi* register has those bits set to zero corresponding to the one bits in the Mask field of the TLB entry.**
- the value returned in the G bit in both the *EntryLo0* and *EntryLo1* registers comes from the single G bit in the TLB entry. Recall that this bit was set from the logical AND of the two G bits in *EntryLo0* and *EntryLo1* when the TLB was written.
- The value returned in the ASID field of the *EntryHi* register is zero for those chips that implement a BAT-based MMU organization.

Restrictions:

This instruction is legal only if the processor is in kernel mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

The operation is UNDEFINED if the contents of the *Index* register are greater than or equal to the number of TLB entries in the processor.

For processors that do not include the standard TLB, the operation of this instruction is **UNDEFINED**.

Operation:

```
i <- Index
if i > TLBEntries -1 then
    UNDEFINED
endif
if (SR_CU0 = 1) or (SR_UM = 0) or (SR_EXL = 1) or (SR_ERL = 1) then
PageMask_Mask <- TLB[i]_Mask
EntryHi <- (TLB[i]_VPN2 and not TLB[i]_Mask) ||
 0^5 || TLB[i]_ASID
EntryLo1 <- TLB[i]_PFN1 || TLB[i]_C1 || TLB[i]_D1 ||
 TLB[i]_V1 || TLB[i]_G
EntryLo0 <- TLB[i]_PFN0 || TLB[i]_C0 || TLB[i]_D0 ||
 TLB[i]_V0 || TLB[i]_G
else
InitiateCoprocessorUnusableException(0)
endif
```

Exceptions:

Coprocessor Unusable Exception

Write Indexed TLB Entry TLBWI

31	26	25	24		0
COP0 010000		CO 1	0 000 0000 0000 0000 0000		TLBWI 000010
6		1	19		6
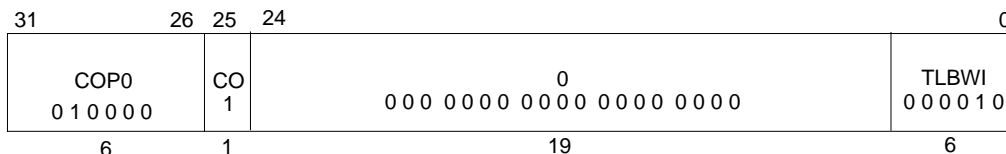
Format:

    TLBWI   MIPS32

Purpose:

Write a TLB entry indexed by the *Index* register.

Description:

The TLB entry pointed to by the *Index* register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1,* and *PageMask* registers. Note that the single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.

Restrictions:

This instruction is legal only if the processor is in kernel mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

The operation is UNDEFINED if the contents of the *Index* register are greater than or equal to the number of TLB entries in the processor.

For processors that do not include the standard TLB, the operation of this instruction is **UNDEFINED**.

Operation:

```
i <- Index
if i > TLBEntries -1 then
UNDEFINED
endif
if (SR_CU0 = 1) or (SR_UM = 0) or (SR_EXL = 1) or (SR_ERL = 1) then
TLB[i]_Mask <- PageMask_Mask
TLB[i]_VPN2 <- EntryHi_VPN2
TLB[i]_ASID <- EntryHi_ASID
TLB[i]_G <- EntryLo1_G and EntryLo0_G
TLB[i]_PFN1 <- EntryLo1_PFN
TLB[i]_C1 <- EntryLo1_C
TLB[i]_D1 <- EntryLo1_D
```

```
            TLB[i]_{V1} <- EntryLo1_V
            TLB[i]_{PFN0} <- EntryLo0_PFN
            TLB[i]_{C0} <- EntryLo0_C
            TLB[i]_{D0} <- EntryLo0_D
            TLB[i]_{V0} <- EntryLo0_V
            else
            InitiateCoprocessorUnusableException(0)
            endif
```

Exceptions:

Coprocessor Unusable Exception

Write Random TLB Entry   TLBWR

```
31          26 25  24                                                0
┌──────────────┬────┬──────────────────────────────────┬──────────────┐
│    COP0      │ CO │              0                   │   TLBWR      │
│   010000     │ 1  │  000 0000 0000 0000 0000          │  000110      │
└──────────────┴────┴──────────────────────────────────┴──────────────┘
       6         1                  19                        6
```

Format:

TLBWR   MIPS32

Purpose:

Write a TLB entry indexed by the *Random* register.

Description:

The TLB entry pointed to by the *Random* register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1,* and *PageMask* registers. Note that the single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.

Restrictions:

This instruction is legal only if the processor is in kernel mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

For processors that do not include the standard TLB MMU, the operation of this instruction is **UNDEFINED**.

Operation:

```
i <- Random
if (SR_CU0 = 1) or (SR_UM = 0) or (SR_EXL = 1) or (SR_ERL = 1) then
TLB[i]_Mask <- PageMask_Mask
TLB[i]_VPN2 <- EntryHi_VPN2
TLB[i]_ASID <- EntryHi_ASID
TLB[i]_G <- EntryLo1_G and EntryLo0_G
TLB[i]_PFN1 <- EntryLo1_PFN
TLB[i]_C1 <- EntryLo1_C
TLB[i]_D1 <- EntryLo1_D
TLB[i]_V1 <- EntryLo1_V
TLB[i]_PFN0 <- EntryLo0_PFN
TLB[i]_C0 <- EntryLo0_C
TLB[i]_D0 <- EntryLo0_D
TLB[i]_V0 <- EntryLo0_V
else
InitiateCoprocessorUnusableException(0)
```
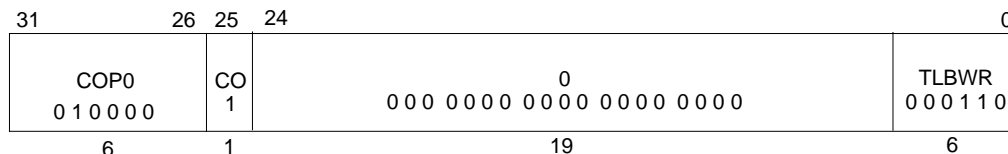
```
                         endif
```

Exceptions:
 Coprocessor Unusable Exception

Trap if Less Than

TLT

31	26	25	21	20	16	15	6	5	0
SPECIAL 0 0 0 0 0 0		rs		rt		code		TLT 1 1 0 0 1 0	
6		5		5		10		6	
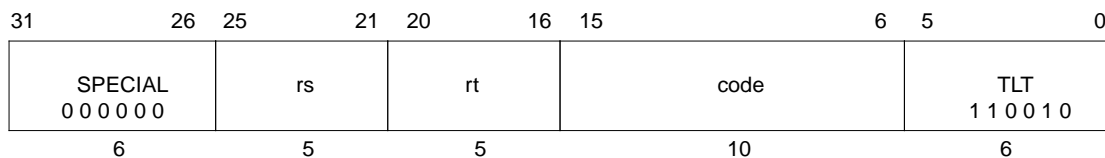
Format: TLT   rs, rt   **MIPS II**

Purpose: To compare GPRs and do a conditional trap

Description: `if rs < rt then Trap`

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is less than GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions: None

Operation:

```
if GPR[rs] < GPR[rt] then
          SignalException(Trap)
          endif
```

Exceptions: Trap

Trap if Less Than Immediate

TLTI

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TLTI 0 1 0 1 0	immediate
6	5	5	16
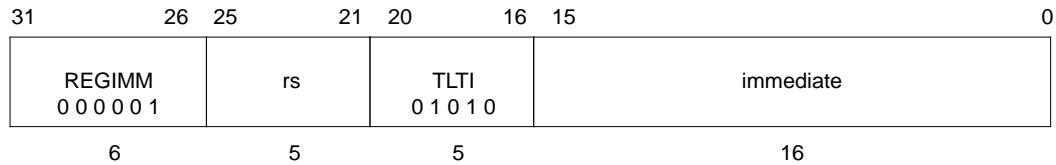
Format: TLTI   rs, immediate   **MIPS II**

Purpose: To compare a GPR to a constant and do a conditional trap

Description: if rs < immediate then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is less than *immediate*, then take a Trap exception.

Restrictions: None

Operation:

```
if GPR[rs] < sign_extend(immediate) then
          SignalException(Trap)
          endif
```

Exceptions: Trap

Trap if Less Than Immediate Unsigned   **TLTIU**

```
 31           26  25           21  20           16  15                                    0
┌─────────────────┬──────────────┬──────────────┬────────────────────────────────────────┐
│    REGIMM       │     rs       │    TLTIU     │              immediate                  │
│    000001       │              │    01011     │                                         │
└─────────────────┴──────────────┴──────────────┴────────────────────────────────────────┘
        6                5              5                         16
```
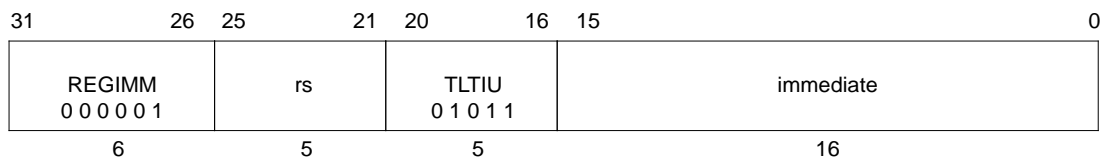
Format: TLTIU   rs, immediate   **MIPS II**

Purpose: To compare a GPR to a constant and do a conditional trap

Description: `if rs < immediate then Trap`

Compare the contents of GPR *rs* and the 16-bit sign-extended *immediate* as unsigned integers; if GPR *rs* is less than *immediate*, then take a Trap exception.

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

Restrictions: None

Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
        SignalException(Trap)
        endif
```

Exceptions: Trap

Trap if Less Than Unsigned   **TLTU**

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TLTU 110011
6	5	5	10	6

Format: TLTU   rs, rt   **MIPS II**

Purpose: To compare GPRs and do a conditional trap

Description: `if rs < rt then Trap`

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; if GPR *rs* is less than GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions: None

Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
          SignalException(Trap)
          endif
```

Exceptions: Trap

Trap if Not Equal **TNE**

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TNE 1 1 0 1 1 0
6	5	5	10	6

Format: TNE rs, rt **MIPS II**

Purpose: To compare GPRs and do a conditional trap

Description: if rs ≠ rt then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is not equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.
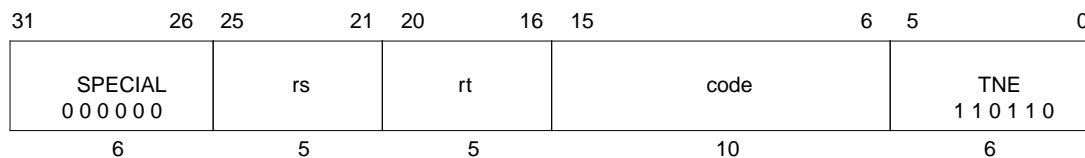
Restrictions: None

Operation:

```
if GPR[rs] ≠ GPR[rt] then
          SignalException(Trap)
          endif
```

Exceptions: Trap

Trap if Not Equal	TNEI

31	26	25	21	20	16	15	0
REGIMM 0 0 0 0 0 1		rs		TNEI 0 1 1 1 0		immediate	
6		5		5		16	

Format: TNEI   rs, immediate   **MIPS II**

Purpose: To compare a GPR to a constant and do a conditional trap

Description: if rs ≠ immediate then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is not equal to *immediate*, then take a Trap exception.

Restrictions: None

Operation:

```
if GPR[rs] ≠ sign_extend(immediate) then
          SignalException(Trap)
          endif
```

Exceptions: Trap

Enter Standby Mode WAIT

31 26	25 24		0
COP0 0 1 0 0 0 0	CO 1	Implementation-Dependent Code	WAIT 1 0 0 0 0 0
6	1	19	6
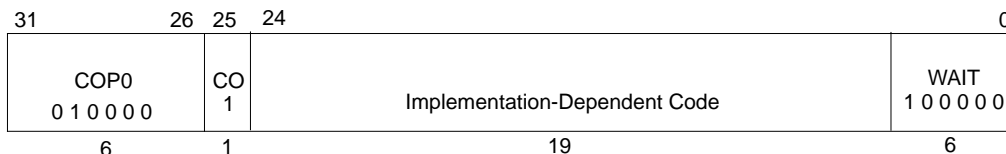
Format:
WAIT MIPS32

Purpose:
Wait for Event

Description:
The WAIT instruction forces the core into low power mode. The pipeline is stalled and when all external requests are completed, the processor's main clock is stopped. The processor will restart when reset (SI_Reset or SI_ColdReset) is signaled, or a non-masked interrupt is taken (SI_NMI, SI_Int, or EJ_DINT). Note that the 4K cores do not use the code field in this instruction.

Restrictions:
The operation of the processor is undefined if a wait instruction is placed in the delay slot of a branch or a jump.

This instruction is legal only if the processor is in kernel mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

Operation:

```
if (SR_CU0 = 1) or (SR_UM = 0) or (SR_EXL = 1) or (SR_ERL = 1) then
    Enter lower power mode
else
    InitiateCoprocessorUnusableException(0)
endif
```
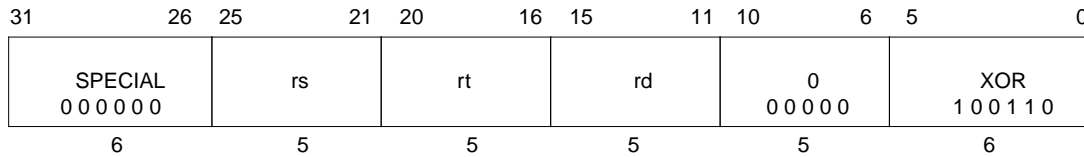
Exceptions:
Coprocessor Unusable Exception

Exclusive OR   XOR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	XOR 100110
6	5	5	5	5	6

Format: XOR   rd, rs, rt   **MIPS I**

Purpose: To do a bitwise logical Exclusive OR

Description: rd ← rs XOR rt

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical Exclusive OR operation and place the result into GPR *rd.*
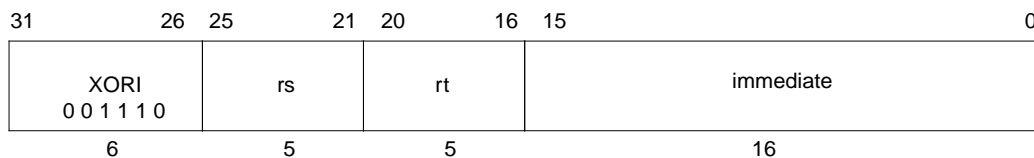
Restrictions: None

Operation:

GPR[rd] ← GPR[rs] xor GPR[rt]

Exceptions: None

Exclusive OR Immediate   **XORI**

```
 31        26  25        21  20        16  15                                        0
┌────────────┬────────────┬────────────┬─────────────────────────────────────────┐
│    XORI    │     rs     │     rt     │                immediate                │
│   001110   │            │            │                                         │
└────────────┴────────────┴────────────┴─────────────────────────────────────────┘
      6             5            5                         16
```

Format: XORI   rt, rs, immediate   **MIPS I**

Purpose: To do a bitwise logical Exclusive OR with a constant

Description: rt ← rs XOR immediate

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

Restrictions: None

Operation:

GPR[rt] ← GPR[rs] xor zero_extend(immediate)

Exceptions: None