MIPS64™ 5K™ Processor Core Family Software User's Manual

Document Number: MD00012
Revision 02.08
May 28, 2002

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

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

MIPS Technologies or any contractually-authorized third party reserves the right to change the information contained in this document to improve function, design or otherwise. MIPS Technologies does not assume any liability arising out of the application or use of this information, or of any error of omission in such information. Any warranties, whether express, statutory, implied or otherwise, including but not limited to the implied warranties of merchantability or fitness for a particular purpose, are excluded. Any license under patent rights or any other intellectual property rights owned by MIPS Technologies or third parties shall be conveyed by MIPS Technologies or any contractually-authorized third party in a separate license agreement between the parties.

The information contained in this document shall not be exported or transferred for the purpose of reexporting in violation of any U.S. or non-U.S. regulation, treaty, Executive Order, law, statute, amendment or supplement thereto.

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

MIPS®, R3000®, R4000®, R5000® and R10000® are among the registered trademarks of MIPS Technologies, Inc. in the United States and certain other countries, and MIPS16™, MIPS16e™,MIPS32™, MIPS64™, MIPS-3D™, MIPS-based™, MIPS I™, MIPS II™, MIPS III™, MIPS IV™, MIPS V™, MDMX™, SmartMIPS™, 4K™, 4Kc™, 4Km™, 4Kp™, 4KE™, 4KEc™, 4KEm™, 4KEp™, 4KS™, 4KSc™, 5K™, 5Kc™, 5Kf™, 20K™, 20Kc™, R20K™, R4300™, ATLAS™, CoreLV™, EC™, JALGO™, MALTA™, MGB™, SEAD™, SEAD-2™, SOC-it™ and YAMON™ are among the trademarks of MIPS Technologies, Inc.

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

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides an introduction to the MIPS Technologies MIPS64™ 5K™ microprocessor core family, with as description of the different members, the 5Kc and 5Kf cores. It contains the following sections:

- Section 1.1, "Overview"
- Section 1.2, "Features"
- Section 1.3, "Core Block Diagram"

1.1 Overview

The MIPS64 5K is a synthesizable, highly-integrated 64-bit MIPS® RISC microprocessor core designed for high-performance, low-power, low-cost embedded applications. The 5K core is portable across processes, is highly configurable, and is integrated easily into standard design flows. It incorporates powerful instructions for embedded applications, as well as proven memory-management and privileged mode control mechanisms.

The 5K core executes the MIPS64™ instruction set architecture (ISA), which is a superset of the MIPS V™ architecture, and includes special multiply-accumulate, conditional move, prefetch, wait, and leading zero/one detect instructions. To allow easy migration from 32-bit processors, the 5K provides a 32-bit compatibility mode, in which code compiled for MIPS32™ processors can run unaltered.

The 5Kf also core also features a high performances IEEE 754 compliant Floating Point Unit (FPU). The FPU supports both single and double precision instructions. It includes the multiply add instruction, which can issue every cycle, whereby both a multiply and an add single precision operation can be performed in every cycle. The 5Kf core can dual issue a floating point arithmetic instruction with a floating point load/store or integer instruction, whereby two instructions can be executed every cycle in floating point applications. A coprocessor interface is also provided, which allows designers a way to easily extend their architectures by addition of custom functionality, such as network, or graphics coprocessors.

The 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, or one 64x64 DMULT/DMULTU every 9 clocks.

The memory management unit contains a configurable 16, 32, or 48 dual-entry Joint TLB (JTLB) with variable page sizes, a 4-entry Instruction micro TLB (ITLB), and a 4-entry Data micro TLB (DTLB). Using a TLB with the 5K core is optional. The alternative is to use a far simpler Fixed Mapping Translation (FMT) scheme.

Optional instruction and data caches are fully programmable from 0 - 64 Kbytes in size, with a maximum size of 16 Kbytes/way in a 4-way, set-associative implementation. In addition, each cache can be organized as direct-mapped, 2-way, 3-way, or 4-way set-associative. The 5K supports an instruction- scheduling mechanism that reduces pipeline stalls on cache misses and also supports hit-under-miss processing in the data cache. Both caches are virtually indexed and physically tagged. Virtual indexing allows the cache to be indexed in the same clock cycle in which the address is translated.

To ease software debugging, the EJTAG debug solution in the 5K core includes instruction software breakpoints, a single-step feature, and a dedicated Debug Mode. Optional hardware breakpoints include 4 instruction and 2 data breakpoints. An optional Test Access Port (TAP) forms the interface to an external debug host and provides a dedicated communication channel for debugging of an embedded system.

1.2 Features

- 64-bit Data and Address Path
  (42-bit virtual and 36-bit physical address space)
- MIPS64 Compatible Instruction Set
  - Based on MIPS V™ instruction set architecture
  - Multiply-accumulate and multiply-subtract instructions (MADD, MADDU, MSUB, MSUBU)
  - Targeted multiply instruction (MUL)
  - Zero/One detect instructions (CLZ, CLO, DLCO, DCLZ)
  - Wait instruction (WAIT)
  - Conditional move instructions (MOVZ, MOVN)
  - Prefetch instructions (PREF, PREFX)
- Dual-issue Floating Point Unit / Coprocessor 1 (5Kf core only)
  - Fully pipelined IEEE 754 compliant floating point unit with both single and double precision instructions
  - Includes multiply add instruction
  - Maximum issue rate of one multiply add single (MADD.S) instruction every clock
  - Maximum issue rate of one multiply add double (MADD.D) instruction every other clock
  - FPU executes independently of integer pipeline
  - Fast flush-to-zero mode to optimize performance
- Dual-issue superscalar micro-architecture capable of executing (5Kf core only):
  - 1 integer and 1 arithmetic floating point instruction
  - 1 floating point arithmetic and 1 floating point load/store instruction
- General Purpose Coprocessor Interface
  - Supports both COP1 and COP2 coprocessors for 5Kc, and COP2 coprocessor for 5Kf
  - Supports all MIPS V instructions, including advanced COP1X instructions for 5Kc
  - Utilizes high-performance features of the integer unit
  - Dual-issue capable interface supports execution of an arithmetic coprocessor instruction, and an integer or coprocessor load/store instruction every cycle
  - Utilizes high-performance features of the integer unit
- Programmable Cache Sizes
  - Individually configurable instruction and data caches
  - Sizes from 0 - 16 KBytes/way (64 KBytes maximum)
  - Direct-mapped, 2-, 3-, or 4-Way Set Associative
  - Non-blocking loads and prefetches
  - 32-byte cache line size, doubleword sectored
  - Virtually indexed, physically tagged
  - Cache line locking support
  - Optional parity protection

- MIPS64 privileged resource architecture
  - Count/Compare registers for real-time timer interrupts
  - Instruction and Data watch registers for software breakpoints
  - Separate interrupt exception vector
  - Supervisor Mode operation
  - Performance Monitoring logic for analyzing application speed
- Programmable Memory Management Unit
  - 16, 32, or 48 dual-entry JTLB with variable page sizes
  - 4-entry instruction micro TLB
  - 4-entry data micro TLB
  - Support for 8-bit ASID
  - Support for 4KB - 16MB page sizes
- Simple Bus Interface Unit (BIU)
  - All I/Os fully registered
  - Separate unidirectional 36-bit address and 64-bit data buses
  - 32-byte write buffer (4 doublewords)
  - 1-line (32-byte) eviction buffer
- Multiply/Divide Unit
  - Max issue rate of one 32x16 multiply per clock
  - Max issue rate of one 32x32 multiply every other clock
  - Max issue rate of one 64x64 multiply every nine clocks
  - 37 clock latency on 32/32 divides.
  - 69 clock latency on 64/64 divides
  - Early-in feature for divides allows results sooner for smaller dividend values
- Power Control
  - Minimum frequency is 0 MHz
  - Power-down mode (triggered by WAIT instruction)
  - Support for software-controlled clock divider
  - Sleep Mode: During this mode, the clocks are shut off. Sleep mode is entered automatically from power-down mode after all bus activity stops.
- EJTAG Debug Support
  - Software Debug Breakpoint instruction (SDBBP)
  - Single-step feature
  - Debug Mode
  - Optional hardware breakpoints (4 instruction and 2 data breakpoints)
  - Optional Test Access Port (TAP) interface to debug host

1.3 Core Block Diagram

The basic blocks that comprise the 5K core are shown for the 5Kc core in Figure 1-1 and for the 5Kf core in Figure 1-2. Blocks that are optional are shown as shaded. The optional blocks can be added to the 5K core, depending on the particular requirements of an implementation.

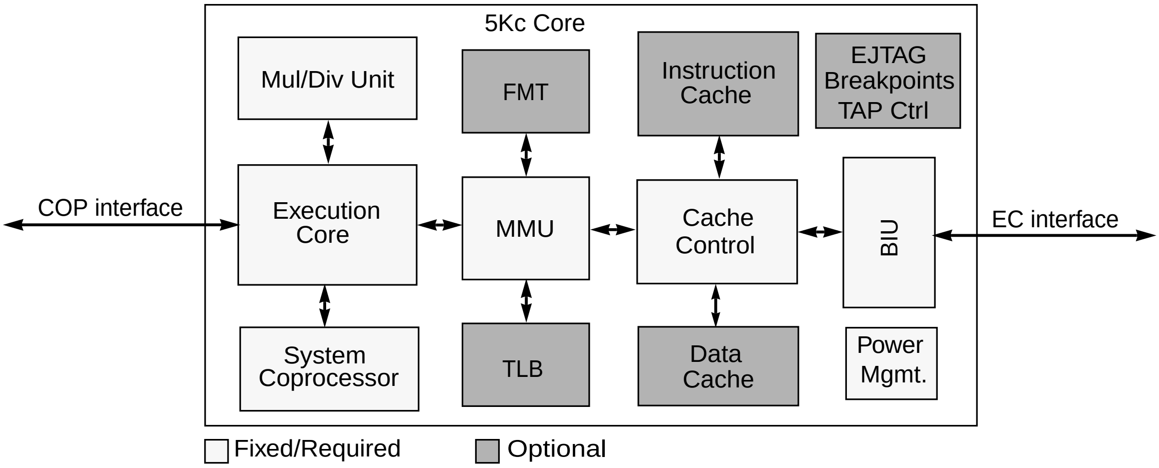


Figure 1-1 5Kc Core Block Diagram

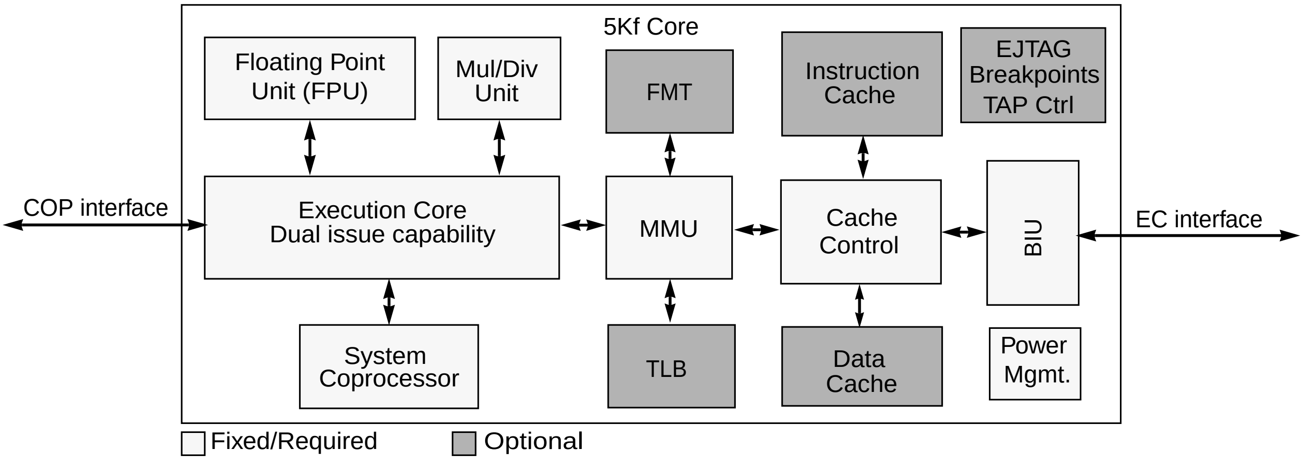


Figure 1-2 5Kf Core Block Diagram

Each block is described individually in the remaining sections of this chapter.

1.3.1 Execution Unit

The 5K core execution unit implements a load/store architecture with single-cycle ALU operations (logical, shift, add, subtract). The 5K core contains thirty-two, 64-bit general-purpose registers used for scalar integer operations and address calculation. The register file consists of two read ports and two write ports and is fully bypassed to minimize operation latency in the pipeline.

The execution unit includes:

- 64-bit adder used for calculating arithmetic results and data addresses

- 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 instructions streams in which data-producing instructions are followed closely by consumers of their results
- Zero/One detect unit for implementing the CLZ, DCLZ and CLO, DCLO instructions
- Logic Unit for performing bitwise logical operations
- Shifter & Store Aligner

1.3.2 Floating Point Unit (FPU) / Coprocessor 1 (5Kf core only)

The 5Kf core Floating Point Unit (FPU) implements the MIPS64 ISA (Instruction Set Architecture) for floating-point computation. The implementation supports the ANSI/IEEE Standard 754 (IEEE Standard for Binary Floating-Point Arithmetic). The hardware supports IEEE single and double precision data formats. The performance is optimized for single precision formats. Most instructions have a 1 cycle throughput and 4 cycle latency. The FPU contains thirty-two 64-bit floating-point registers used for floating point operations.

The FPU implements the MIPS64 multiply-add (MADD) and multiply-sub (MSUB) instructions with intermediate rounding after the multiply function. The result is guaranteed to be the same as executing a MUL and an ADD instruction separately, but the instruction latency, instruction fetch, dispatch bandwidth, and the total number of register accesses are improved. A fast Flush-to-Zero mode is implemented to optimize performance. IEEE denormalized input operands are supported by hardware for some instructions. IEEE denormalized result operands are not supported by hardware.

The FPU has a separate pipeline for floating point instruction execution. This pipeline operates in parallel with the integer unit (IU) pipeline and does not stall when the IU pipeline stalls. This allows long-running FPU operations, such as divides or square root, to be partially masked by system stalls and/or other integer unit instructions. Instructions are always dispatched and completed in order. The exception model is ‘precise’ at all times. The FPU is also denoted coprocessor 1.

For additional information, refer to Chapter 3, “Floating-Point Unit.”

1.3.3 Multiply/Divide Unit (MDU)

The Multiply/Divide unit performs multiply and divide operations. The MDU supports execution of a 16x16 or 32x16 multiply operation every clock cycle. 32x32 multiply operations can be issued every other clock cycle, and 64x64 multiply operations can be issued every nine clock cycles. Appropriate interlocks are implemented to stall the issue of back-to-back 32x32 and 64x64 multiply operations. Multiply operand size is automatically determined by logic in the MDU.

The MDU contains 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, multicycle MDU operations (such as a divide) to be partially masked from system stalls and other integer unit instructions. To achieve the highest possible performance, the MDU contains a 32x16 Booth-recoded multiply array, and each class of multiply instructions is processed in a different way, so as to best utilize available resources.

Divide operations are implemented with a simple 1 bit-per-clock iterative algorithm. A 32-bit divide requires 37 clock cycles to complete, while a 64-bit divide requires 69 clock cycles. 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. However, the divider has an early-in feature which detects the size of the dividend in 8-bit increments, so that when a smaller dividend is detected, the algorithm reduces the number of iterations accordingly.

For additional information, refer to Chapter 2, “Pipeline.”

1.3.4 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 (for example, interrupts enabled or disabled), the operating modes (Kernel, Supervisor, User, or Debug Mode), and the processor’s diagnostics capability. Configuration information such as cache size and set associativity is available by accessing the CP0 registers.

For additional information, refer to Chapter 6, “Coprocessor 0 Registers.”

1.3.5 Memory Management Unit (MMU)

The 5K core contains an MMU that interfaces between the execution unit and the cache controller. The MMU supports two types of address translation mechanisms, either of which can be implemented:

- Translation lookaside buffer (TLB)
- Fixed Mapping Translation (FMT)

The TLB consists of three address translation buffers, a 16, 32, or 48 dual-entry fully associative Joint TLB (JTLB), a 4-entry fully associative Instruction TLB (ITLB), and a 4-entry fully associative Data TLB (DTLB).

For additional information, refer to Chapter 4, “Memory Management.”

1.3.6 Cache Controllers & Bus Interface

The instruction and data cache controllers support caches of various sizes, organizations, and set-associativity. For example, the data cache can be 8 Kbytes in size and 2-way set-associative, while the instruction cache can be 16 Kbytes in size and 4-way set-associative. In addition, each cache has its own 64-bit data path, and both caches can be accessed in the same pipeline clock cycle.

The Bus Interface Unit (BIU) controls the external interface signals.

For additional information, refer to Chapter 8, “Cache Organization and Operation.”

1.3.7 Power Management

The 5K microprocessor 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 system that execution and clocking should be halted, thereby reducing system power consumption during idle periods.

For additional information, refer to Chapter 9, “Power Management.”

1.3.8 Instruction and Data Caches

The 5K core supports optional, on-chip instruction and data caches that can each be accessed in a single processor cycle. The caches are 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.

While cache refills are in progress, the caches can continue processing hits. Streaming is also supported, in which instructions and data are forwarded during cache refills. Cache performance is further enhanced by special logic that implements a least-recently used (LRU) algorithm for way selection when a cache line is replaced.

The 5K core supports instruction cache locking. This feature allows critical code and data segments to be locked into the cache, enabling the system designer to maximize the efficiency of the system cache. For the data cache, the locked contents can be updated on a store hit, but cannot be selected for replacement on a store miss. Cache locking is always enabled on all cache entries—entries can be marked as locked or unlocked on a per-line basis using the CACHE instruction.

For additional information, refer to Chapter 8, "Cache Organization and Operation."

1.3.9 EJTAG Debug Support

The 5K core provides an optional Enhanced JTAG (EJTAG) interface for use in software debugging of application and operating-system code. In addition to standard User, Supervisor, and Kernel Modes, the 5K core provides a Debug Mode which is entered after a Debug exception is taken and continues until a Debug Exception Return (DERET) instruction is executed.

For additional information, refer to Chapter 5, "Exception Processing," and Chapter 10, "EJTAG Debug Features."

Chapter 2

Pipeline

Chapter 2 describes the 5K processor core instruction pipeline. The pipeline includes a six-stage integer pipeline and a separate execution pipeline for multiply and divide operations. The two pipelines operate in parallel.

This chapter contains the following sections:

- Section 2.1, "Pipeline Stages"
- Section 2.2, "Instruction Fetch"
- Section 2.3, "Branch Delay"
- Section 2.4, "Limited Dual Issue"
- Section 2.5, "Instruction Fetching from Uncached Memory Space"
- Section 2.6, "Data Access"
- Section 2.7, "Instruction Scheduling"
- Section 2.8, "MDU Pipeline"
- Section 2.9, "Slip Conditions and Interlock Handling"

2.1 Pipeline Stages

The integer pipeline consists of the following six stages:

- Instruction Fetch (I Stage)
- Instruction Dispatch (D Stage)
- Register File Read (R Stage)
- Instruction Execution (E Stage)
- Memory Access (M Stage)
- Writeback (W stage)

The 5K core implements 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.

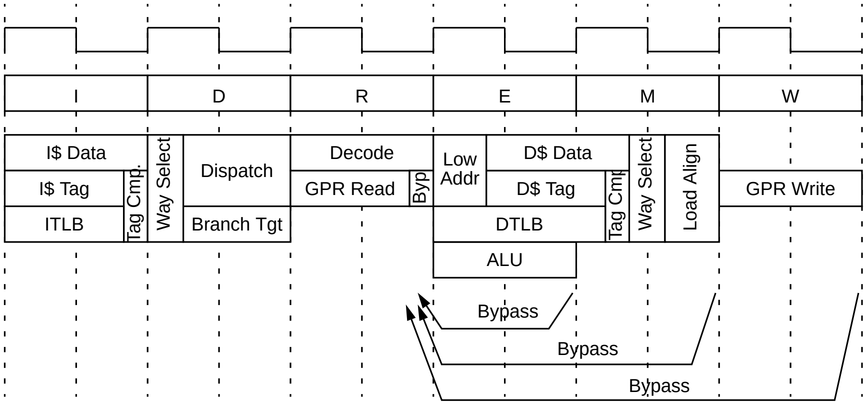


Figure 2-1 Pipeline Stages

2.1.1 I Stage: Instruction Fetch

During the I stage:

- Instruction(s) are fetched from the instruction cache.
- The instruction translation lookaside buffer (ITLB) performs the virtual-to-physical address translation.

2.1.2 D Stage: Instruction Dispatch

During the D stage:

- Branch decode and prediction.
- Instruction dispatch to coprocessor/integer unit.

2.1.3 R Stage: Register File Read

During the R stage:

- The GPR register file is read.
- The instruction is decoded.

2.1.4 E Stage: Execution

During the E stage:

- The Arithmetic Logic Unit (ALU) performs the arithmetic or logical operation for register-to-register instructions.
- The ALU determines whether the branch condition is true.
- All multiply and divide operations begin in this stage.
- The ALU calculates the full virtual address for load and store instructions.
- The cache look-up starts for loads and stores.

2.1.5 M Stage: Memory Access

During the M stage:

- The data translation lookaside buffer (DTLB) performs the virtual-to-physical address translation.
- Data cache look-up completes.
- Loaded data is aligned.

2.1.6 W Stage: Writeback

During the W stage:

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

2.2 Instruction Fetch

The 5K processor maintains a three-entry doubleword instruction buffer, which can store up to six instructions. The instruction buffer includes the following features:

- Speculative fetching of several instructions.
- Intelligent handling of instruction cache misses.
- Minimization of penalty for branches and jumps.

Using speculative instruction fetching, the instruction buffer can keep ahead of the rest of the pipeline by prefetching instructions which may be dispatched at a later time. Instruction-cache refills are potentially initiated early, thus minimizing delays in the pipeline.

To avoid unnecessary instruction-cache refills, the instruction buffer employs a conservative, intelligent cache-refill scheme, ensuring that a cache line is refilled only if it contains an instruction which is certain to be dispatched to the execution pipeline. For example, speculative refills are performed only if there are no jumps or branches in the instruction buffer.

When a branch instruction is recognized, the instruction buffer ensures that one entry contains the branch target and one entry contains the instruction after the branch delay slot. The instruction buffer predicts all branch instructions to be taken. Correctly predicted branches impose no pipeline delay. When the branch is mispredicted (not taken), the total penalty will be only one pipeline bubble (as explained in Section 2.3, "Branch Delay").

Note that the JR and JALR instructions always cause one pipeline bubble because of the interlock resulting from a data dependency on the source register.

When required by the instruction buffer, the instruction cache is accessed. The instruction address is translated to determine if the required instruction resides in the cache. If it does not, an instruction-cache miss occurs.

When a cache miss is detected, the fetch request is issued on the external bus. If the instruction buffer is empty, the I stage is slipped (refer to Section 2.9, "Slip Conditions and Interlock Handling"). When the instruction is returned, the I-stage slip is released, and the instruction is written to the instruction register for immediate use.

2.3 Branch Delay

The 5K pipeline has a branch delay of one cycle. The one-cycle delay allows a branch target address calculated during the D stage to be used for the instruction access in the following I stage. Use of a branch delay slot avoids a pipeline bubble on all correctly-predicted branch instructions.

The pipeline begins fetching the predicted path in the cycle following the delay slot. The branch condition is calculated in the E stage of the branch. If the condition of the branch was mispredicted, the correct path is followed one cycle later, causing a bubble in the pipeline. Note that at this point the mispredicted path will have been already speculatively fetched.

Figure 2-2 shows a correctly-predicted branch. A mispredicted branch is shown in Figure 2-3.

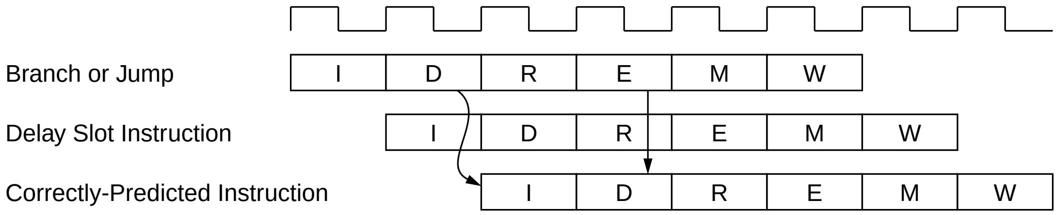
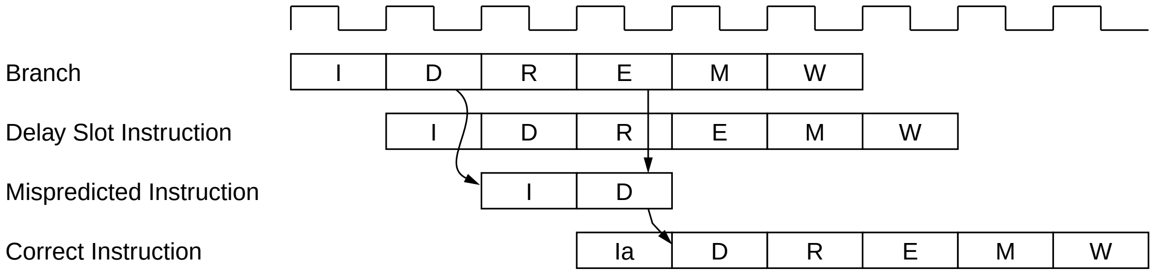


Figure 2-2 Correctly-predicted Branch



a. Speculatively fetched before mispredicted instruction

Figure 2-3 Mispredicted Branch

Keep in mind that the instruction buffer has some limitations, namely, that a stall in the pipeline may cause a later stall because of the nature of the instruction cache system. This occurs in two cases:

- When a taken branch or jump at an even address jumps to an instruction at an odd address, the branch or jump must be stalled in its D stage. This stall causes a stall of one clock cycle in the I stage of the branch or jump target.
- When a not-taken branch at an even address is stalled in its R stage. This stall causes a stall of one clock cycle in the I stage of the instruction following the delay slot of the branch.

These two cases are shown in Figure 2-4 and Figure 2-5. Figure 2-4 shows a taken branch or jump at an even address that jumps to an odd address, causing a stall in the D stage of the branch or jump. Figure 2-5 shows a not-taken branch at an even address, which causes a stall in the branch’s R stage.

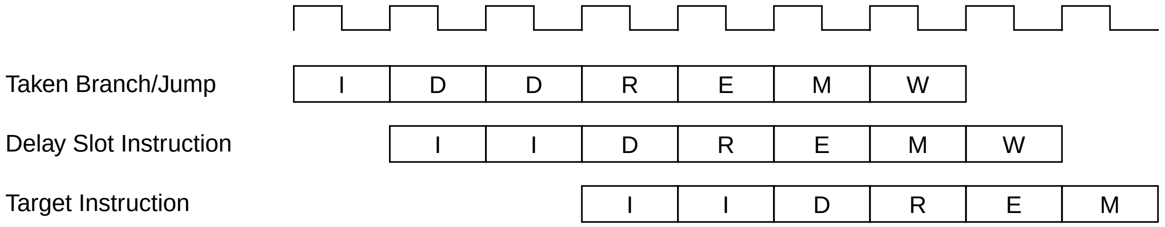


Figure 2-4 Taken Branch/Jump at Even Address Jumps to Instruction at Odd Address

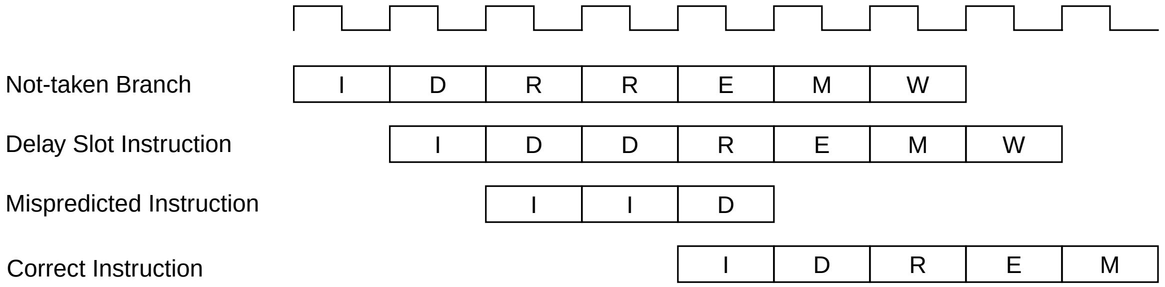


Figure 2-5 Not-taken Branch At Even Address

2.4 Limited Dual Issue

The 5K processor employs a performance-enhancing dual issue dispatch scheme, known as “Limited Dual Issue”. Whenever possible, so-called “arithmetic” coprocessor instructions will be dispatched in parallel with instructions to the integer pipeline. By writing code for the coprocessor(s) with this in mind, it is possible to achieve a performance of up to two instructions per clock cycle.

Below instructions are grouped as either dual issueable arithmetic coprocessor instructions or dual issueable non-arithmetic instructions. Instructions not in one of these groups will always be single issued.

Table 2-1 Arithmetic Coprocessor Instructions which can be Dual Issued

Major Opcodes	Minor Opcodes[a]
MDMX	ALL Except ALNV.fmt
COP1	IR[25] == 1 Except MOVN.fmt, MOVZ.fmt
COP1X	IR[5:4] != 00 Except ALNV.PS
COP2	IR[25] == 1

[a] IR[x] refers to bit or bit range x in the instruction opcode

Table 2-2 Non-Arithmetic Instructions which can be Dual Issued

Major Opcodes	Minor Opcodes	Additional Restrictions: Not dual issued when FR bit in CP0 Status register is zero
ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI, LUI DADDI, DADDIU, LDL, LDR LB, LH, LWL, LW, LBU LHU, LWR, LWU SB, SH, SWL, SW, SDL, SDR, SWR, CACHE LL, LWC1, LWC2, PREF, LLD, LDC1, LDC2, LD SC, SWC1, SWC2, SCD, SDC1, SDC2, SD	N/A	LL, LWC1, LWC2, PREF, LLD, LDC1, LDC2, LD SC, SWC1, SWC2, SCD, SDC1, SDC2, SD
SPECIAL	All with following exceptions: SSNOP (subset of SLL) MOVCI JR JALR	No restrictions
COP1	MFC1, DMFC1, MTC1, DMTC1	Not dual issued
COP1X	LWXC1, LDXC1, LUXC1 SWXC1, SDXC1, SUXC1 PREFX	Not dual issued
COP2	MFC2, DMFC2, MTC2, DMTC2	No restrictions

Note that the above tables excludes dual issue of (among other instructions) SSNOP, all branches and jumps, ERET, DERET, MOVCI, CTCx/CFCx and MTC0/MFC0. Furthermore, an instruction in a branch/jump delay slot will never be dual issued.

When an instruction pair consisting of one arithmetic and one non-arithmetic is present in the instruction buffers, the 5K processor will try to dual issue the instruction pair if both instructions are included in Table 2-1 and Table 2-2. However, the ability to dual issue instructions degrades just after jumps/branches.

Following rules of thumb should be employed to ensure the best dual issue performance of code containing arithmetic instructions:

- Interleave arithmetic and non-arithmetic instructions
- Double word align the instruction pairs which are to be dual issued (unaligned instruction pairs can be dual issued but the instruction buffers are better utilized when instructions are aligned)
- Limit the number of branches and jumps

Following instruction sequence is an example of floating-point code which significantly utilizes the dual-issue capability of the 5K processor. The code displayed is the manually optimized inner loop of a Mandelbrot application.

```
loop:   madd.s  fp14, fp18, fp10, fp10
        sltu    r2, r8, r9
        msub.s  fp15, fp18, fp11, fp11
        mfc1    r12, fp17
        mul.s   fp16, fp10, fp11
        mfc1    r22, fp27
        madd.s  fp24, fp28, fp20, fp20
        sltu    r11, r12, r4
        msub.s  fp25, fp28, fp21, fp21
        sltu    r21, r22, r4
        add.s   fp17, fp14, fp15
        or      r3, r11, r21
```

```
mul.s   fp26, fp20, fp21
and     r2, r3, r2
sub.s   fp10, fp14, fp15
movn    r20, r8, r21
add.s   fp27, fp24, fp25
movn    r10, r8, r11
madd.s  fp11, fp13, fp2, fp16
addiu   r8, r8, 1
sub.s   fp20, fp24, fp25
bne     r2, r0, loop
madd.s  fp21, fp23, fp2, fp26
```

Note the following:

- Floating point arithmetic instructions (MADD.s, MSUB.s, MUL.s, SUB.s, ADD.s) are interleaved with non-arithmetic instructions (SLTU, MFC1, OR, AND, MOVN, ADDIU).
- Data dependencies between instructions are allowed to resolve with no stalls by rearranging the code. An FPU instruction normally have a latency of 4 clock cycles, so in order to avoid stalls the producer (instruction producing result) and the consumer (instruction using result) must be separated by 3 clock cycles which equivalents to 6 dual issued instructions.
- Above loop of 23 instructions executes in 14 clock cycles. Theoretically it could execute in 12 clock cycles but dual issue possibilities are missed due to the branch (delay slot and branch target).
- Each loop iteration computes two pixels in the Mandebrot picture. By computing two pixels in parallel it is possible to avoid the FPU to idle due to dependency stalls. The calculation speed is thus 7 clock cycles per pixel per iteration. The performance can be increased further by unrolling the loop, i.e. minimizing the number of branches.

2.5 Instruction Fetching from Uncached Memory Space

The MIPS ISA prohibits speculative fetches from uncached memory space, that is, only those instructions that are certain to be dispatched to the pipeline may be speculatively fetched. To comply with this rule, one pair of doubleword-aligned instructions is fetched at a time, and all dependencies for the two instructions are resolved. Only when the fetch unit has determined that neither instruction is a jump or branch and that no exception occurred during their execution, the next instruction pair is fetched. This results in the pipeline performance shown in Figure 2-6.

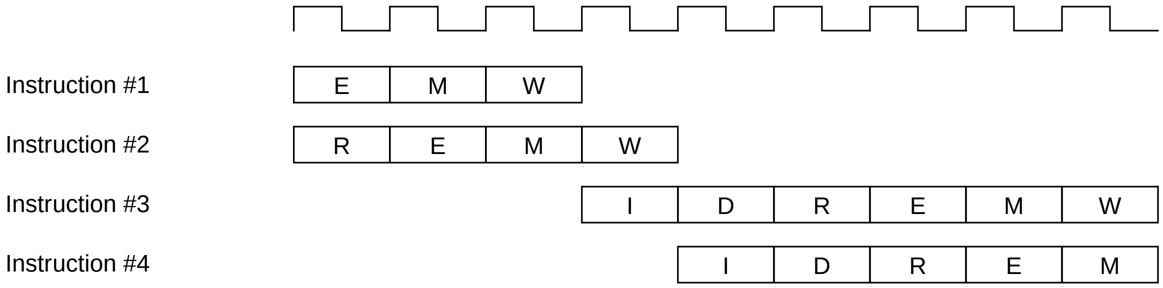


Figure 2-6 Instruction Fetching from Uncached Memory Space

In configurations without an instruction cache, this restrictive fetching scheme can be avoided, so that pipeline performance is limited only by the bandwidth to external memory. To enable this feature, specify the memory pages as cacheable (even though no cache exists). Speculative fetching and full usage of the instruction buffer will then occur within those memory pages. For information on specifying the memory attributes of pages, refer to Chapter 4, “Memory Management.”

2.6 Data Access

As explained in Chapter 8, “Cache Organization and Operation,” when a data access is requested, the data address is translated to determine if the required data resides in the cache. If it does not, a data-cache miss occurs.

When a data-cache miss is detected in the M stage, the core may slip the M stage until the miss has been resolved and the requested word becomes available (in case it is a load). The number of clocks required to return the data is determined by the access time on the external bus. However, in many cases the 5K processor core can continue executing other instructions while fetching the load data on the external bus, thus avoiding any slips caused by the data-cache miss. This feature is described in Section 2.7, "Instruction Scheduling".

When the required data is returned, it passes through the aligner before being forwarded to the execution unit and register file.

2.7 Instruction Scheduling

For some multicycle instructions, the 5K processor core is able to write results to the register file out-of-order, allowing other instructions to execute while an earlier instruction is waiting to deliver its data. This feature is called *instruction scheduling*, that is, an instruction is scheduled to write-back to the register file at a later time (for load instructions, this feature is known as a *non-blocking cache)*.

The instructions that can be scheduled are: LB, LBU, LH, LHU, LW, LWU, LD, MFHI, MFLO, MUL, MFC1, DMFC1, CFC1, MFC2, DMFC2 and CFC2. Only one scheduled instruction can be outstanding at a time.

While an instruction is scheduled, other instructions can execute freely; however, once the data is needed by an instruction, that instruction slips its R stage until the data is returned. Other schedulable instructions can execute while an instruction is scheduled. Loads and stores that hit in the cache can execute while a previous load is scheduled (a feature called *hit-under-miss*).

Because only one instruction can be scheduled at a time, the M stage of a second schedulable instruction is slipped until it delivers its data or until the previous scheduled instruction delivers its data, thereby causing the second instruction to become scheduled. For example, two loads that both miss in the cache will cause the M stage of the second load to slip until the first load returns its data.

In order to optimally utilize the instruction-scheduling feature, schedulable instructions should be placed as far as possible from the instruction that uses the data, thus maximizing the number of instructions that can execute while waiting for the data.

2.8 MDU Pipeline

The 5K processor core contains 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 integer pipeline stalls. This allows multicycle MDU operations to be partially masked from system stalls and other integer unit instructions. To achieve the highest possible performance, the MDU contains a 32x16 Booth-recoded multiply array, and each class of multiply instructions is processed in a different way, so as to best utilize available resources.

2.8.1 Multiply/MAC Operations

The MDU supports all of the MIPS64 multiply operations. Operands can be 16, 32, or 64 bits in size. For all multiply and multiply-accumulate operations, the MDU dynamically determines the size of the *rt* operand, with allowable operations of 32x16, 32x32, and 64x64.

Because the MIPS64 architecture defines its general-purpose registers to be 64 bits, precise definitions are needed for the terms *16-bit operand*, *32-bit operand*, and *64-bit operand*.

An operand is a 16-bit operand if either of the following conditions is true:

1. The operation is signed and the upper 48 bits are equal to bit 15.
2. The operation is unsigned and the upper 48 bits are 0.

An operand is a 32-bit operand if it is not a 16-bit operand as described above, and either of the following is true:

1. The operation is signed and the upper 32 bits are equal to bit 31.
2. The operation is unsigned and the upper 32 bits are 0.

If an operand does not qualify as a 16-bit or 32-bit operand, then it is a 64-bit operand. (Note that the results of single-word multiply instructions are not defined if either of the operands are 64 bits.)

Given the above definitions, the three types of operations—32x16, 32x32, and 64x64—can also be precisely defined:

A multiply is defined as a 32x16 multiply if both of the following are true:

- The *rs* operand is 16 bits or 32 bits.
- The *rt* operand is 16 bits.

A multiply is defined as a 32x32 multiply if both of the following are true:

- The *rs* operand is 16 bits or 32 bits.
- The *rt* operand is 32 bits.

A multiply is defined to be a 64x64 multiply only for the DMULT/DMULTU operations, and only if it is not a 32x16 or 32x32 multiply, as defined above.

Figure 2-7 shows a diagram of a 32x16 multiply operation. In the first cycle, the *rs* and *rt* operands arrive and the Booth recoding function occurs. The multiply array requires one clock and occurs in the second cycle. For MAC operations, the accumulation also occurs in the second cycle. In the third cycle, the carry-propagate-add (CPA) function occurs and the operation completes.

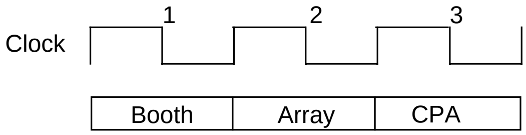


Figure 2-7 MDU Pipeline Flow During a 32x16 Multiply Operation

Figure 2-8 shows a diagram of a 32x32 multiply operation. In the first cycle, the *rs* and *rt* operands arrive and the Booth recoding function is performed. The multiply array, requiring two clocks, occurs in the second and third cycles. For multiply-accumulate (MAC) operations, the accumulation also occurs in the second cycle. Booth recoding is performed in the second cycle for the second pass through the array. In the fourth cycle, the CPA function occurs and the operation completes.

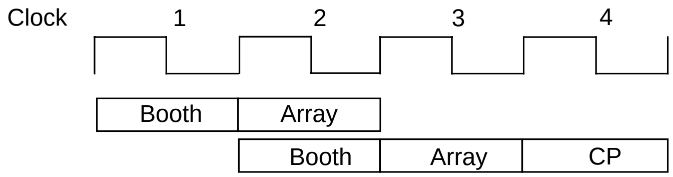


Figure 2-8 MDU Pipeline Flow During a 32x32 Multiply Operation

Figure 2-9 shows a diagram of a 64x64 multiply operation. It requires eight passes through the array and an additional CPA cycle, for a total of nine cycles. Booth recoding is always performed in the cycle before the pass through the array. The CPA function is used multiple times to accumulate intermediate results into the final HI/LO results.

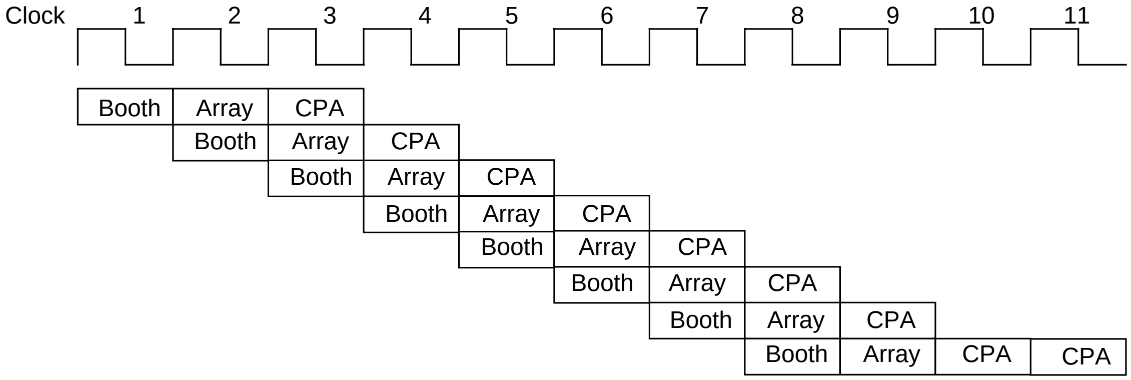


Figure 2-9 MDU Pipeline Flow During a 64x64 Multiply Operation

2.8.2 Divide Operations

Divide operations are implemented with a simple, 1-bit-per-clock nonrestoring division algorithm. Thus, for DIV/DIVU instructions, 32 cycles are required to complete the algorithm, and for DDIV/DDIVU instructions, 64 cycles are required. In order to speed up this algorithm, logic is included which detects small values on the *rs* (dividend). Thus, if *rs* is actually an 8-bit value, only 8 iterations are required. This logic detects 8-bit, 16-bit, 24-bit, 32-bit, 40-bit, 48-bit, and 56-bit *rs* values. To complete the calculation, some additional cycles are required, as described below.

Because the nonrestoring division algorithm can only be used for positive operands, an initialization cycle is required to negate the *rs* operand. To eliminate critical timing paths and simplify the logic, this negation cycle is taken even for positive *rs* operands, although the negation itself is not performed.

One cycle is then used to detect small *rs* operands, as described above. If a small *rs* operand is detected, then the starting dividend is shifted the required number of bits in this cycle.

At the end of the computation, a negative remainder may result. If so, a final iteration is required to make the remainder positive. If the final remainder is positive, this extra iteration is zeroed out and does not affect the final result. After this final iteration, the quotient and remainder need to be sign-extended and possibly negated if the division results in a negative number.

Figure 2-10 shows a diagram for a 32-bit divide operation. As shown in the figure, the DIV/DIVU instructions require up to 37 cycles to complete: one cycle for initialization, once cycle for shifting operands, up to 32 cycles of iteration, one final iteration, and two cycles for quotient and remainder negation.

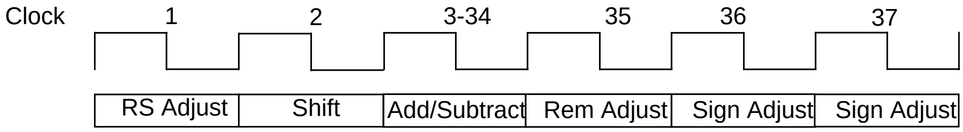


Figure 2-10 MDU Pipeline Flow During a 32-bit Divide Operation

Figure 2-11 shows a diagram of a 64-bit divide operation. As shown in the figure, DDIV/DDIVU require up to 69 cycles to complete: one cycle of initialization, one cycle for shifting operands, up to 64 cycles of iteration, one final iteration, and two cycles for quotient and remainder negation.

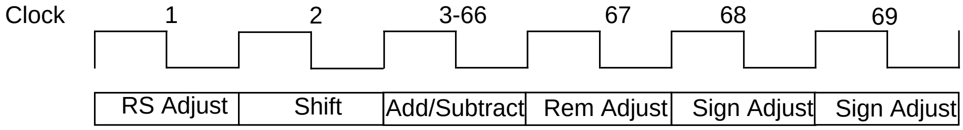


Figure 2-11 MDU Pipeline Flow During a 64-bit Divide Operation

2.8.3 Latencies and Repeat Rates

Table 2-3 shows the number of cycles required by the MDU to compute the result for each of the multiply and divide instructions. Because the MDU can sometimes bypass results before they are written back, this table does not describe pipeline interlocks. For a complete description of when MDU instructions interlock, refer to Table 2-6 on page 23.

Table 2-3 5K Core Instruction Latencies

Operation	Instructions	Latency (in clock cycles)
32x16 Multiply/MAC	DMULT/DMULTU, MULT/MULTU, MADD/MADDU, MSUB/MSUBU, MUL	3
32x32 Multiply/MAC	DMULT/DMULTU, MULT/MULTU, MADD/MADDU, MSUB/MSUBU, MUL	4
64x64 Multiply	DMULT/DMULTU	11
64/64 Divide	DDIV/DDIVU	69
56/64 Divide	DDIV/DDIVU	61
48/64 Divide	DDIV/DDIVU	53
40/64 Divide	DDIV/DDIVU	45
32/32 or 32/64 Divide	DIV/DIVU DDIV/DDIVU	37
24/32 or 24/64 Divide	DIV/DIVU DDIV/DDIVU	29
16/32 or 16/64 Divide	DIV/DIVU DDIV/DDIVU	21

Table 2-3 5K Core Instruction Latencies (Continued)

Operation	Instructions	Latency (in clock cycles)
8/32 or 8/64 Divide	DIV/DIVU DDIV/DDIVU	13

Table 2-4 shows the repeat rates for multiply instructions. These numbers are of interest only for those instructions that normally repeat, namely, multiply-accumulate instructions and the MUL instruction.

Table 2-4 5K Core Instruction Repeat Rates

Operation	Instructions	Repeat Rate (in clock cycles)
32x16 MAC	MADD/MADDU, MSUB/MSUBU	every cycle
32x32 MAC	MADD/MADDU, MSUB/MSUBU	every other cycle
32x16 Multiply	MUL	every cycle
32x32 Multiply	MUL	every other cycle

The MDU implements hardware interlocking to stall the integer pipeline if it is busy when another MDU operation is dispatched by the integer unit. Because of this interlocking, there are no software restrictions on using the MDU instructions.

2.8.4 MDU Interaction with Integer Unit Pipeline

MDU operations begin when an instruction enters the E-stage of the integer pipeline, after which the MDU pipeline runs more or less independently from the integer pipeline. For instructions that write a result back to the integer register file (MFHI, MFLO, MUL), the MDU will send the results during the M-stage of the integer pipeline, if they are available. If they are not available, the integer unit will schedule these instructions.

Figure 2-12 shows the pipeline flow for the following sequence of instructions:

1. 32x16 multiply (M_1)
2. ADD
3. 32x32 multiply (M_2)

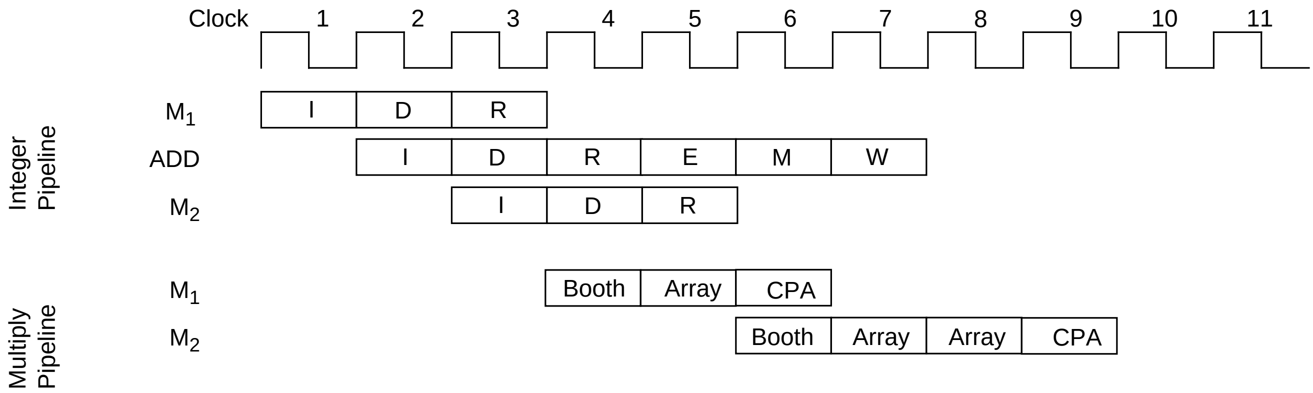


Figure 2-12 Integer Pipeline and MDU Pipeline Interaction

A clock-by-clock analysis of Figure 2-12 is described below.

Clock 1: The first 32x16 multiply operation, M_1, enters the I stage and is fetched from the instruction cache.

Clock 2: M_1 enters the D stage, and the ADD operation enters the I stage.

Clock 3: M_1 enters the R stage. The ADD operation enters the D stage. The 32x32 multiply operation, M_2, enters the I stage and is fetched from the instruction cache.

Clock 4: M_1 is passed to the Booth stage of the MDU pipeline. M_2 enters the D stage. In this clock cycle, there is no activity in the E stage of the integer pipeline.

Clock 5: M_1 enters the Array stage. The ADD operation enters the E stage of the integer pipeline. M_2 enters the R stage.

Clock 6: M_1 passes to the CPA stage of the MDU pipeline (because M_1 is a 32x16 multiply operation, only one clock is required for the Array stage). The ADD operation enters the M stage of the integer pipeline. M_2 enters the Booth stage.

Clock 7: M_1 completes and its result is written to the HI/LO register pair. The ADD operation completes and is written to the register file in the W stage of the integer pipeline. M_2 enters the Array stage.

Clock 8: Because a 32x32 multiply requires two passes through the multiplier, M_2 remains in the Array stage.

Clock 9: M_2 enters the CPA stage of the MDU pipeline

Clock 10: M2 completes and its result is written to the HI/LO register pair at the end of the clock cycle.

2.9 Slip Conditions and Interlock Handling

An interlock is a condition which halts the smooth flow of the pipeline, for example, a cache miss in the M stage. Interlocks are resolved by hardware—in each clock cycle, interlock conditions are checked for all active instructions, and the condition is handled, for example, by a cache refill operation.

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

Table 2-5 Pipeline Interlocks

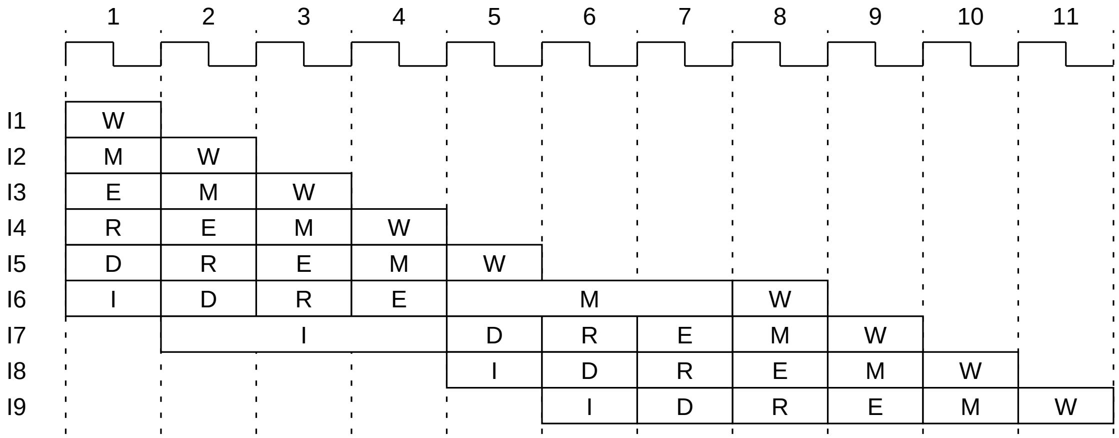
Interlock Type	Sources	Slip Stage
ITLB Miss	Instruction TLB	I Stage
Instruction cache miss	Instruction cache	I Stage
Instruction	See Table 2-6	
DTLB Miss	Data TLB	M Stage
Data cache miss	Unscheduled load that misses in data cache	M Stage
	Multicycle cache operation	
	SYNC or CACHE instruction	
	Store when write-through buffer is full	
	Store hit in cache refill buffer	

Processors designed by MIPS Technologies, Inc. resolve hardware interlocks either by stalling the pipeline, that is, stopping all instructions in all stages (also called *freezing* the pipeline), or by using a *slip*, in which only a part of the pipeline is held static, while other parts of the pipeline can continue to advance. In the 5K core processor, all interlocks are resolved by a slip—in every clock cycle, internal logic determines whether each pipe stage is allowed to advance.

Slip conditions may propagate to preceding stages; for example, when the M stage does not advance, the E stage of the next instruction cannot advance either. However, if the next instruction has not yet entered its E stage, it can advance, thus eliminating any potential bubbles in the pipeline.

Slipped instructions are retried on subsequent cycles, as preceding pipeline stages advance and perhaps resolve the dependency. Hardware inserts NOPs in the bubbles caused by the slip.

Figure 2-13 shows examples of pipeline slips.



	1	2	3	4	5	6	7	8	9	10	11
I1	W										
I2	M	W									
I3	E	M	W								
I4	R	E	M	W							
I5	D	R	E	M	W						
I6	I	D	R	E	M			W			
I7		I			D	R	E	M	W		
I8					I	D	R	E	M	W	
I9						I	D	R	E	M	W

Figure 2-13 Pipeline Slip

In Figure 2-13, there is two-cycle slip in the M stage and in the I stage of two consecutive instructions. In the second clock cycle, the pipeline is full, and an instruction-cache miss is detected for I7. The instruction-cache miss occurs in clock cycle 2, when the I7 instruction fetch is attempted. In this example, two clock cycles (3 and 4) are required to fetch the I7 instruction from memory. Note that during this time, the previous instructions (I1, I2, I3, I4, I5, and I6) can advance and/or complete in the pipeline; however, I8 and I9 cannot start. After the cache miss is resolved in clock cycle 4 and the first word of I7 is written to the cache, the pipeline is restarted, causing I7 instruction to advance from the I stage to the D stage in clock 5.

In the fifth clock cycle, a DTLB miss is detected for I6, which causes a two-cycle slip of the M stage. During this time, instructions I7, I8, and I9, can continue to advance in the pipeline, eliminating any possible bubbles caused by the I cache miss on I7.

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

Table 2-6 Instruction Interlocks

First Instruction		Following Instruction(s)	Issue Delay (in Clock Cycles)	Slip Stage
Any		JR or JALR without any data dependency on previous instruction	1	D stage
Any		JR or JALR that consumes data from previous instruction	2	D/R stage
LB/LBU/LH/LHU/LL/LW/LWL/LWR, LWU/LLD/LD/LDL/LDR, MFC0/DMFC0/MFC1/DMFC1/CFC1, MFC2/DMFC2/CFC2, MFHI/MFLO/MUL, SC/SCD		JR or JALR that consumes data from previous instruction	3	D/R stage
LB/LBU/LH/ LHU/LL/LW/ LWL/LWR, LWU/LLD/LD/LDL/LDR		Instruction that consumes load data	1	R stage
MFC0/DMFC0/MFC1/DMFC1/CFC1, MFC2/DMFC2/CFC2		Instruction that consumes data in a target register	1	R stage
Any		TLBR	1	M stage
TLBWR/TLBWI		TLBWR/TLBWI TLBR/TLBP	2	M stage
TLBWR/TLBWI		Any instruction accessing JTLB (i.e., I-fetch, load, store, PREF, PREFX, CACHE miss in ITLB or DTLB)	2	M stage
SC/SCD		Consumer of fail/success status	1	R stage
MFHI/MFLO		Consumer of destination register	1	R stage
MULT/MULTU DMULT/ DMULTU MADD/ MADDU MSUB/ MSUBU	32x16	MFLO/MFHI followed by N instructions not using destination register followed by consumer of destination register[a]	2-N	R stage
	32x32		3-N	
	64x64		10-N	
MUL	32x16	N instructions not using destination register followed by consumer of destination register	3-N	R stage
	32x32		4-N	

Table 2-6 Instruction Interlocks (Continued)

First Instruction		Following Instruction(s)	Issue Delay (in Clock Cycles)	Slip Stage
MULT/MULTU MUL MADD/MADDU MSUB/MSUBU	32x16	MULT/MULTU DMULT/DMULTU MUL/MADD/MADDU MSUB/MSUBU MTHI/MTLO	0	R stage
	32x32		1	
DMULT/DMULTU	32x16	MULT/MULTU DMULT/DMULTU MUL	0	R stage
	32x32		1	
	64x64		8	
DMULT/DMULTU	32x16	MADD/MADDU MSUB/MSUBU MTHI/MTLO	1	R stage
	32x32		2	
	64x64		9	
MTHI/MTLO		DIV/DIVU/DDIV/DDIVU	2	R stage
MULT/MULTU DMULT/DMULTU MUL MADD/MADDU MSUB/MSUBU	32x16	DIV/DIVU/DDIV/DDIVU	2	R stage
	32x32		3	
	64x64		11	
DIV/DIVU DDIV/DDIVU	8/32 or 8/64	MFLO/MFHI followed by N instructions not using destination register followed by consumer of destination register[a]	12-N	R stage
	16/32 or 16/64		20-N	R stage
	24/32 or 24/64		28-N	R stage
	32/32 or 32/64		36-N	R stage
DDIV/DDIVU	40/64	MFLO/MFHI followed by N instructions not using destination register followed by consumer of destination register[a]	44-N	R stage
	48/64		52-N	R stage
	56/64		60-N	R stage
	64/64		68-N	R stage
DIV/DIVU DDIV/DDIVU	8/32 or 8/64	MULT/MULTU DMULT/DMULTU MUL	10	R stage
	16/32 or 16/64		18	R stage
	24/32 or 24/64		26	R stage
	32/32 or 32/64		34	R stage
DDIV/DDIVU	40/64	MULT/MULTU DMULT/DMULTU MUL	42	R stage
	48/64		50	R stage
	56/64		58	R stage
	64/64		66	R stage

Table 2-6 Instruction Interlocks (Continued)

First Instruction		Following Instruction(s)	Issue Delay (in Clock Cycles)	Slip Stage
DIV/DIVU DDIV/DDIVU	8/32 or 8/64	MADD/MADDU MSUB/MSUBU MTHI/MTLO	11	R stage
	16/32 or 16/64		19	R stage
	24/32 or 24/64		27	R stage
	32/32 or 32/64		35	R stage
DDIV/DDIVU	40/64	MADD/MADDU MSUB/MSUBU MTHI/MTLO	43	R stage
	48/64		51	R stage
	56/64		59	R stage
	64/64		67	R stage
DIV/DIVU DDIV/DDIVU	8/32 or 8/64	DIV/DIVU DDIV/DDIVU	12	R stage
	16/32 or 16/64		20	R stage
	24/32 or 24/64		28	R stage
	32/32 or 32/64		36	R stage
DDIV/DDIVU	40/64	DIV/DIVU DDIV/DDIVU	44	R stage
	48/64		52	R stage
	56/64		60	R stage
	64/64		68	R stage

a. If a multiply or divide instruction is in progress, MFHI/MFLO instructions cannot immediately return the result. This does not cause a slip in the pipeline unless a subsequent instruction uses the result from the MFHI/MFLO.

Chapter 3

Floating-Point Unit

This chapter describes the MIPS64 Floating-Point Unit (FPU) included in the 5Kf core. This chapter contains the following sections:

3.1 Features Overview

The FPU is provided via Coprocessor 1. Together with its dedicated system software, the FPU fully complies with the ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*. The MIPS architecture supports the recommendations of IEEE Standard 754, and the coprocessor implements a precise exception model. The key features of the FPU are listed below:

- Full 64-bit operation is implemented in both the register file and functional units.
- A 32-bit Floating-Point Control Register controls the operation of the FPU, and monitors condition codes and exception conditions.
- Like the main processor core, Coprocessor 1 is programmed and operated using a Load/Store instruction set. The processor core communicates with Coprocessor 1 using a dedicated coprocessor interface. The FPU functions as an autonomous unit. The hardware is completely interlocked such that, when writing software, the programmer does not have to worry about inserting delay slots after loads and between dependent instructions.
- The FPU can dual issue arithmetic and load/store instructions, whereby arithmetic operations can operate continuously while data is provided and retrieved. Details about dual issuing are provided in Section 2.4, "Limited Dual Issue".
- Additional arithmetic operations not specified by IEEE Standard 754 (for example, reciprocal and reciprocal square root) are specified by the MIPS architecture and are implemented by the FPU. In order to achieve low latency counts, these instructions satisfy more relaxed precision requirements.
- The MIPS architecture further specifies compound multiply-add instructions. These instructions meet the IEEE accuracy specification where the result is numerically identical to an equivalent computation using multiply, add, subtract, or negate instructions.

Figure 3-1 depicts a block diagram of the FPU.

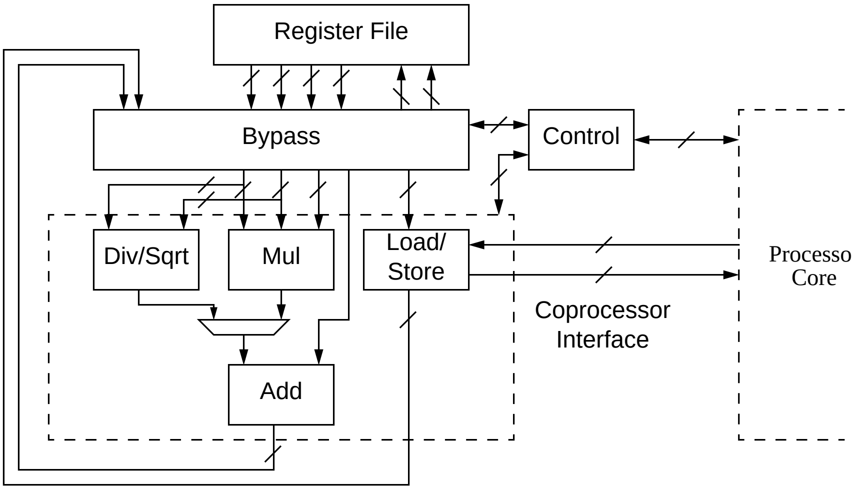


Figure 3-1 FPU Block Diagram

The MIPS architecture is designed such that a combination of hardware and software can be used to implement the architecture. The 5Kf core FPU can operate on numbers within a specific range (in general, the IEEE normalized numbers), but it relies on a software handler to operate on numbers not handled by the FPU hardware (in general, the IEEE denormalized numbers). Supported number ranges for different instructions are described later in this chapter. A fast Flush To Zero mode is provided to optimize performance for cases where IEEE denormalized operands and results are not supported by hardware. The fast Flush to Zero mode is enabled through the CP1 *FCSR* register; use of this mode is recommended for best performance.

3.1.1 IEEE Standard 754

The IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, is referred to in this chapter as “IEEE Standard 754”. IEEE Standard 754 defines the following:

- Floating-point data types
- The basic arithmetic, comparison, and conversion operations
- A computational model

IEEE Standard 754 does not define specific processing resources nor does it define an instruction set.

For more information about this standard, see the IEEE web page at `http://stdsbbs.ieee.org/`.

3.2 Enabling the Floating-Point Coprocessor

Coprocessor 1 is enabled through the CU1 bit in the CP0 *Status* register. When Coprocessor 1 is not enabled, any attempt to execute a floating-point instruction causes a Coprocessor Unusable exception.

3.3 Data Formats

The FPU provides both floating-point and fixed-point data types, which are described below:

- The single- and double-precision floating-point data types are those specified by IEEE Standard 754.
- The fixed-point types are signed integers provided by the CPU architecture.

3.3.1 Floating-Point Formats

The FPU provides the following two floating-point formats:

- a 32-bit single-precision floating point (type S, shown in Figure 3-2)
- a 64-bit double-precision floating point (type D, shown in Figure 3-3)

The floating-point data types represent numeric values as well as the following special entities:

- Two infinities, +∞ and -∞
- Signaling non-numbers (SNaNs)
- Quiet non-numbers (QNaNs)
- Numbers of the form: $(-1)^s 2^E b_0.b_1 b_2..b_{p-1}$, where:
  - s = 0 or 1
  - E = any integer between E_min and E_max, inclusive
  - b_i = 0 or 1 (the high bit, b_0, is to the left of the binary point)
  - p is the signed-magnitude precision

The single and double floating-point data types are composed of three fields—sign, exponent, fraction—whose sizes are listed in Table 3-1.

Table 3-1 Parameters of Floating-Point Data Types

Parameter	Single	Double
Bits of mantissa precision, p	24	53
Maximum exponent, E_max	+127	+1023
Minimum exponent, E_min	-126	-1022
Exponent *bias*	+127	+1023
Bits in exponent field, *e*	8	11
Representation of b_0 integer bit	hidden	hidden
Bits in fraction field, *f*	23	52
Total format width in bits	32	64
Magnitude of largest representable number	3.4028234664e+38	1.7976931349e+308
Magnitude of smallest normalized representable number	1.1754943508e-38	2.2250738585e-308

Layouts of these three fields are shown in Figures 3-2 and 3-3 below. The fields are:

- 1-bit sign, *s*
- Biased exponent, *e = E + bias*
- Binary fraction, $f = .b_1 b_2..b_{p-1}$ (the b_0 bit is *hidden*; it is not recorded)

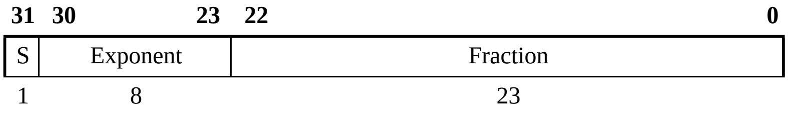


Figure 3-2 Single-Precision Floating-Point Format (S)

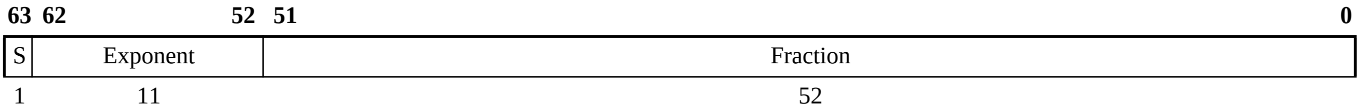


Figure 3-3 Double-Precision Floating-Point Format (D)

Values are encoded in the specified format using the unbiased exponent, fraction, and sign values listed in Table 3-2. The high-order bit of the Fraction field, identified as b_1, is also important for NaNs.

Table 3-2 Value of Single or Double Floating-Point Data Type Encoding

Unbiased E	f	s	b_1	Value V	Type of Value	Typical Single Bit Pattern[a]	Typical Double Bit Pattern[a]
E_max + 1	≠ 0		1	SNaN	Signaling NaN	`0x7fffffff`	`0x7fffffff ffffffff`
			0	QNaN	Quiet NaN	`0x7fbfffff`	`0x7ff7ffff ffffffff`
E_max +1	0	1		- ∞	Minus infinity	`0xff800000`	`0xfff00000 00000000`
		0		+ ∞	Plus infinity	`0x7f800000`	`0x7ff00000 00000000`

E_max
to
E_min
		1		$-\ (2^{E})(1.f)$	Negative normalized number	`0x80800000`
						`through`
						`0xff7fffff`
							`0x80100000 00000000`
							`through`
							`0xffefffff ffffffff`

		0		$+\ (2^{E})(1.f)$	Positive normalized number	`0x00800000`
						`through`
						`0x7f7fffff`
							`0x00100000 00000000`
							`through`
							`0x7fefffff ffffffff`

E_min -1	≠ 0	1		$-\ (2^{E\_min})(0.f)$	Negative denormalized number	`0x807fffff`	`0x800fffff ffffffff`
		0		$+\ (2^{E\_min})(0.f)$	Positive denormalized number	`0x007fffff`	`0x000fffff ffffffff`
E_min -1	0	1		- 0	Negative zero	`0x80000000`	`0x80000000 00000000`
		0		+ 0	positive zero	`0x00000000`	`0x00000000 00000000`

[a] The “Typical” nature of the bit patterns for the NaN and denormalized values reflects the fact that the sign might have either value (NaN) and that the fraction field might have any non-zero value (both). As such, the bit patterns shown are one value in a class of potential values that represent these special values.

3.3.1.1 Normalized and Denormalized Numbers

For single and double data types, each representable nonzero numerical value has just one encoding; numbers are kept in normalized form. The high-order bit of the p-bit mantissa, which lies to the left of the binary point, is “hidden,” and not recorded in the *Fraction* field. The encoding rules permit the value of this bit to be determined by looking at the value of the exponent. When the unbiased exponent is in the range *E_min* to *E_max*, inclusive, the number is normalized and the hidden bit must be 1. If the numeric value cannot be normalized because the exponent would be less than *E_min*, then the representation is denormalized, the encoded number has an exponent of *E_min* – 1, and the hidden bit has the value 0. Plus and minus zero are special cases that are not regarded as denormalized values.

3.3.1.2 Reserved Operand Values—Infinity and NaN

A floating-point operation can signal IEEE exception conditions, such as those caused by uninitialized variables, violations of mathematical rules, or results that cannot be represented. If a program does not trap IEEE exception conditions, a computation that encounters any of these conditions proceeds without trapping but generates a result indicating that an exceptional condition arose during the computation. To permit this case, each floating-point format defines representations (listed in Table 3-2) for plus infinity (+∞), minus infinity (-∞), quiet non-numbers (QNaN), and signaling non-numbers (SNaN).

3.3.1.3 Infinity and Beyond

Infinity represents a number with magnitude too large to be represented in the given format; it represents a magnitude overflow during a computation. A correctly signed ∞ is generated as the default result in division by zero operations and some cases of overflow as described in Section 3.7.2, "Exception Conditions".

Once created as a default result, ∞ can become an operand in a subsequent operation. The infinities are interpreted such that -∞ < (every finite number) < +∞. Arithmetic with ∞ is the limiting case of real arithmetic with operands of arbitrarily large magnitude, when such limits exist. In these cases, arithmetic on ∞ is regarded as exact, and exception conditions do not arise. The out-of-range indication represented by ∞ is propagated through subsequent computations. For some cases, there is no meaningful limiting case in real arithmetic for operands of ∞. These cases raise the Invalid Operation exception condition as described in Section 3.7.2.1, "Invalid Operation Exception".

3.3.1.4 Signalling Non-Number (SNaN)

SNaN operands cause an Invalid Operation exception for arithmetic operations. SNaNs are useful values to put in uninitialized variables. An SNaN is never produced as a result value.

IEEE Standard 754 states that “Whether copying a signaling NaN without a change of format signals the Invalid Operation exception is the implementor’s option.” The MIPS architecture makes the formatted operand move instructions (MOV.fmt, MOVT.fmt, MOVF.fmt, MOVN.fmt, MOVZ.fmt) non-arithmetic; they do not signal IEEE 754 exceptions.

3.3.1.5 Quiet Non-Number (QNaN)

QNaNs provide retrospective diagnostic information inherited from invalid or unavailable data and results. Propagation of the diagnostic information requires information contained in a QNaN to be preserved through arithmetic operations and floating-point format conversions.

QNaN operands do not cause arithmetic operations to signal an exception. When a floating-point result is to be delivered, a QNaN operand causes an arithmetic operation to supply a QNaN result. When possible, this QNaN result is one[1] of the operand QNaN values. QNaNs do have effects similar to SNaNs on operations that do not deliver a floating-point result—specifically, comparisons. (For more information, see the detailed description of the floating-point compare instruction, C.cond.fmt.).

When certain invalid operations not involving QNaN operands are performed but do not trap (because the trap is not enabled), a new QNaN value is created. Table 3-3 shows the QNaN value generated when no input operand QNaN value can be copied. The values listed for the fixed-point formats are the values supplied to satisfy IEEE Standard 754 when a QNaN or infinite floating-point value is converted to fixed point. There is no other feature of the architecture that detects or makes use of these “integer QNaN” values.

[1] In case of one or more QNaN operands, a QNaN is propagated from one of the operands according to the following priority: 1: fs, 2: ft, 3: fr.

Table 3-3 Value Supplied When a New Quiet NaN is Created

Format	New QNaN value
Single floating point	`0x7fbf ffff`
Double floating point	`0x7ff7 ffff ffff ffff`
Word fixed point	`0x7fff ffff`
Longword fixed point	`0x7fff ffff ffff ffff`

3.3.2 Fixed-Point Formats

The FPU provides two fixed-point data types:

- a 32-bit Word fixed point (type W), shown in Figure 3-4
- a 64-bit Longword fixed point (type L), shown in Figure 3-5

The fixed-point values are held in 2's complement format, which is used for signed integers in the CPU. Unsigned fixed-point data types are not provided by the architecture; application software can synthesize computations for unsigned integers from the existing instructions and data types.

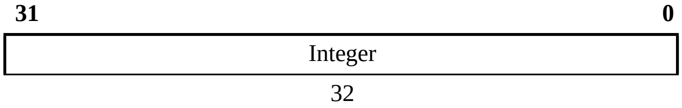


Figure 3-4 Word Fixed-Point Format (W)

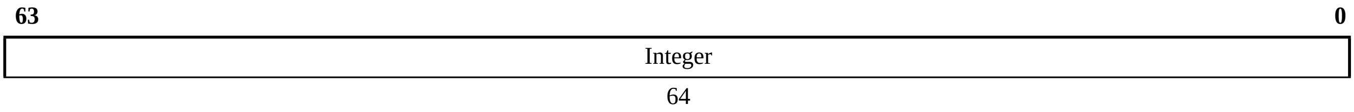


Figure 3-5 Longword Fixed-Point Format (L)

3.4 Floating-Point General Registers

This section describes the organization and use of the Floating-Point general Registers (FPRs). To support MIPS32 programs, the MIPS64 5Kf processor core also provides the MIPS32 register model. The FR bit in the CP0 *Status* register determines which mode is selected:

- When the FR bit is a 1, the MIPS64 register model is selected, which defines 32 64-bit registers with all formats supported in a register.
- When the FR bit is a 0, the MIPS32 register model is selected, which defines 32 32-bit registers with D-format values stored in even-odd pairs of registers; thus the register file can also be viewed as having 16 64-bit registers.

These registers transfer binary data between the FPU and the system, and are also used to hold formatted FPU operand values.

3.4.1 FPRs and Formatted Operand Layout

FPU instructions that operate on formatted operand values specify the Floating-Point Register (FPR) that holds the value. Operands that are only 32 bits wide (*W* and *S* formats) use only half the space in an FPR.

Figures 3-6 and 3-7 show the FPR organization and the way that operand data is stored in them.

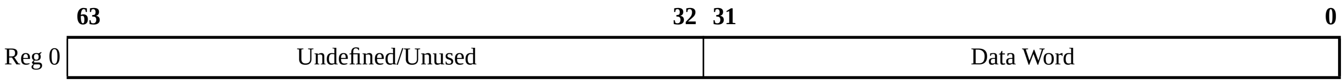


Figure 3-6 Single Floating-Point or Word Fixed-Point Operand in an FPR

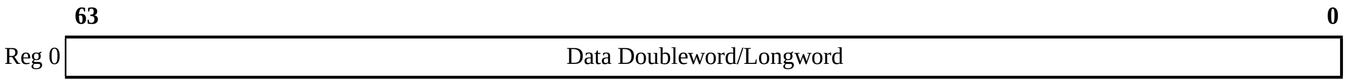


Figure 3-7 Double Floating-Point or Longword Fixed-Point Operand in an FPR

3.4.2 Formats of Values Used in FP Registers

Unlike the CPU, the FPU neither interprets the binary encoding of source operands nor produces a binary encoding of results for every operation. The value held in a floating-point operand register (FPR) has a format, or type, and it can be used only by instructions that operate on that format. The format of a value is either *uninterpreted*, *unknown*, or one of the valid numeric formats: *single* or *double* floating point, and *word* or *long* fixed point.

The value in an FPR is always set when a value is written to the register as follows:

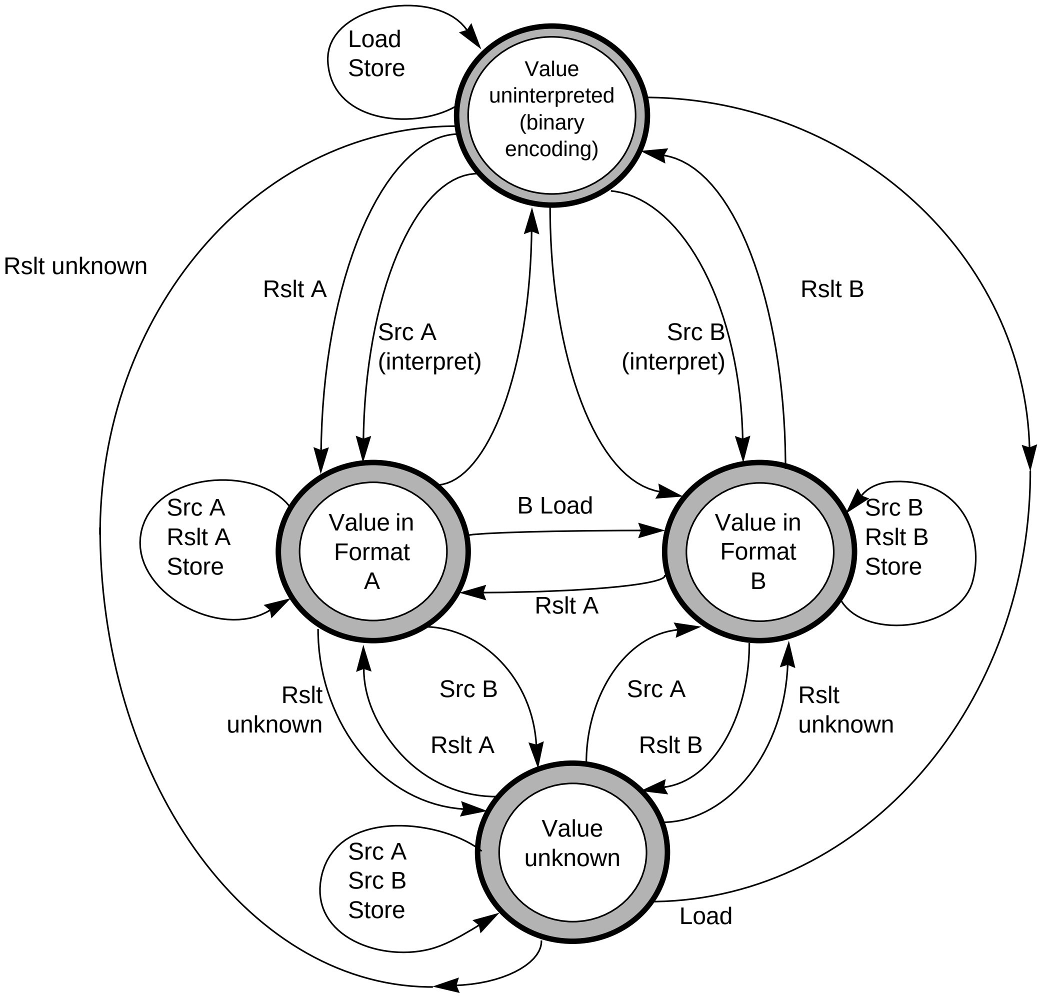
- When a data transfer instruction writes binary data into an FPR (a load), the FPR receives a binary value that is *uninterpreted*.
- A computational or FP register move instruction that produces a result of type *fmt* puts a value of type *fmt* into the result register.

When an FPR with an *uninterpreted* value is used as a source operand by an instruction that requires a value of format *fmt*, the binary contents are interpreted as an encoded value in format *fmt*, and the value in the FPR changes to a value of format *fmt*. The binary contents cannot be reinterpreted in a different format.

If an FPR contains a value of format *fmt*, a computational instruction must not use the FPR as a source operand of a different format. If this case occurs, the value in the register becomes *unknown*, and the result of the instruction is also a value that is *unknown*. Using an FPR containing an *unknown* value as a source operand produces a result that has an *unknown* value.

The format of the value in the FPR is unchanged when it is read by a data transfer instruction (a store). A data transfer instruction produces a binary encoding of the value contained in the FPR. If the value in the FPR is *unknown*, the encoded binary value produced by the operation is not defined.

The state diagram in Figure 3-8 illustrates the manner in which the formatted value in an FPR is set and changed.



A, B: Example formats
Load: Destination of LWC1, LDC1, MTC1, or DMTC1 instructions.
Store: Source operand of SWC1, SDC1, MFC1, or DMFC1 instructions.
Src fmt: Source operand of computational instruction expecting format “fmt.”
Rslt fmt: Result of computational instruction producing value of format “fmt.”

Figure 3-8 Effect of FPU Operations on the Format of Values Held in FPRs

3.4.3 Binary Data Transfers (32-Bit and 64-Bit)

The data transfer instructions move words and doublewords between the FPU FPRs and the remainder of the system. The operations of the word and doubleword load and move-to instructions are shown in Figure 3-9 and Figure 3-10, respectively.

The store and move-from instructions operate in reverse, reading data from the location that the corresponding load or move-to instruction had written.

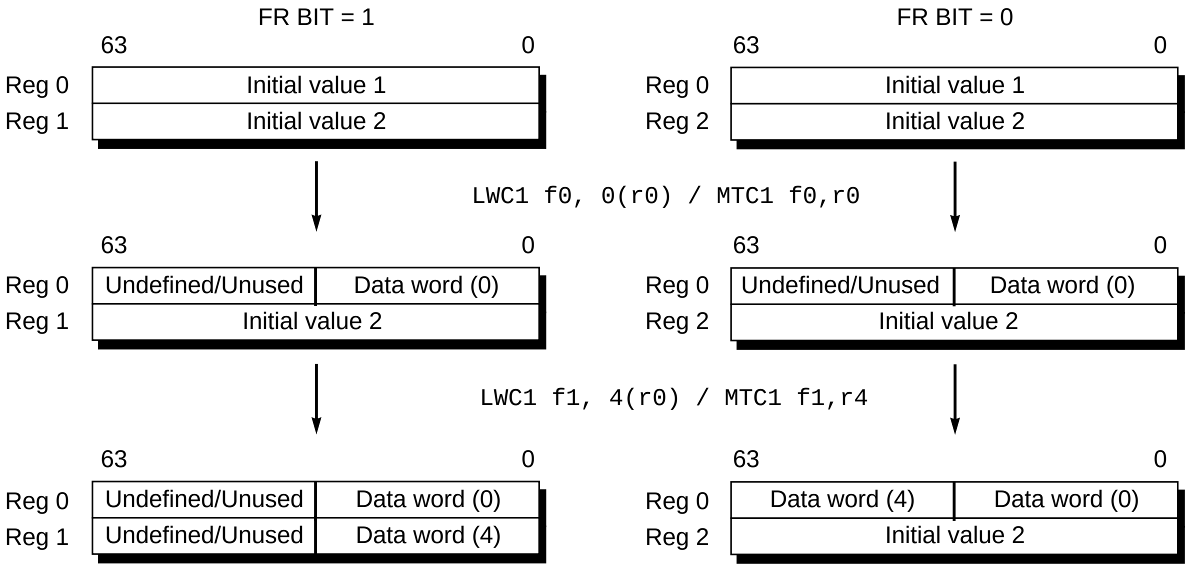


Figure 3-9 FPU Word Load and Move-to Operations

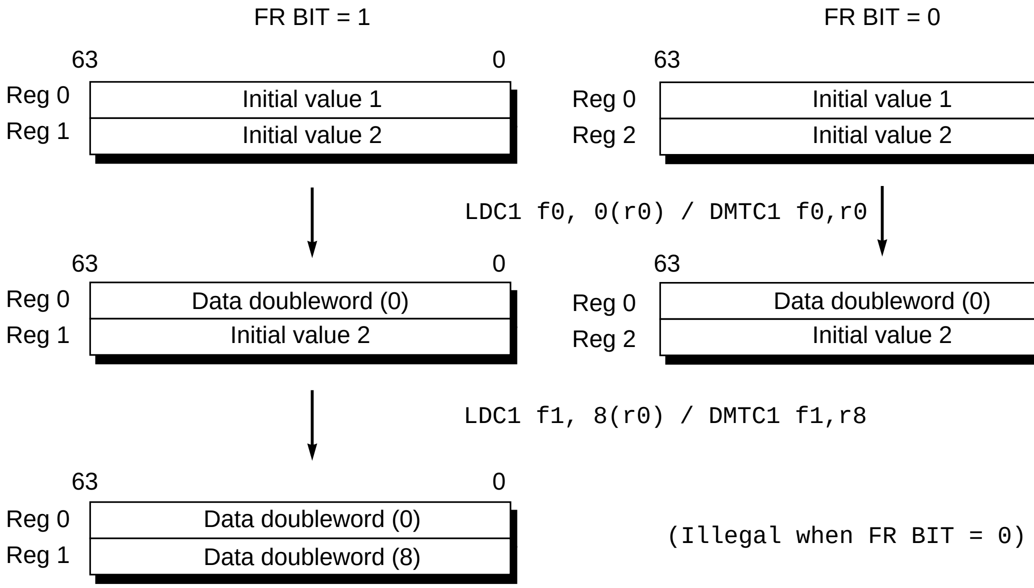


Figure 3-10 FPU Doubleword Load and Move-to Operations

3.5 Floating-Point Control Registers

The FPU Control Registers (FCRs) identify and control the FPU. The five FPU control registers are 32 bits wide: *FIR*, *FCCR*, *FEXR*, *FENR*, *FCSR*. Three of these registers, *FCCR*, *FEXR*, and *FENR*, select subsets of the floating-point Control/Status register, the *FCSR*. These registers are also denoted Coprocessor 1 (CP1) control registers.

CP1 control registers are summarized in Table 3-4 and are described individually in the following subsections of this chapter. Each register's description includes the read/write properties and the reset state of each field.

Table 3-4 Coprocessor 1 Register Summary

Register Number	Register Name	Function
0	FIR	Floating-Point Implementation register. Contains information that identifies the FPU.
25	FCCR	Floating-Point Condition Codes register.
26	FEXR	Floating-Point Exceptions register.
28	FENR	Floating-Point Enables register.
31	FCSR	Floating-Point Control and Status register.

Table 3-5 defines the notation used for the read/write properties of the register bit fields.

Table 3-5 Read/Write Properties

<table>
<tr><th>Read/Write Notation</th><th>Hardware Interpretation</th><th>Software Interpretation</th></tr>
<tr><td>R/W</td><td colspan="2">All bits in this field are readable and writable by software and potentially by hardware.
Hardware updates of this field are visible by software reads. Software updates of this field are visible by hardware reads.
If the reset state of this field is “Undefined,” either software or hardware must initialize the value before the first read returns a predictable value. This definition should not be confused with the formal definition of UNDEFINED behavior.</td></tr>
<tr><td>R</td><td>This field 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.</td><td>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.</td></tr>
<tr><td>0</td><td>Hardware does not update this field. Hardware can assume a zero value.</td><td>The value software writes to this field must be zero. Software writes of non-zero values to this field might 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.</td></tr>
</table>

3.5.1 Floating-Point Implementation Register (FIR, CP1 Control Register 0)

The Floating-Point Implementation Register (*FIR*) is a 32-bit read-only register that contains information identifying the capabilities of the FPU, the Floating-Point processor identification, and the revision level of the FPU. Figure 3-11 shows the format of the *FIR*; Table 3-6 describes the *FIR* bit fields.

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="12">0</td><td>3D</td><td>PS</td><td>D</td><td>S</td><td colspan="8">ProcessorID</td><td colspan="8">Revision</td></tr>
</table>

Figure 3-11 FIR Format

Table 3-6 FIR Bit Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
3D	19	Indicates that the MIPS-3D ASE is implemented: 0: MIPS-3D not implemented 1: MIPS-3D implemented This bit is always 0 to indicate that MIPS-3D is not implemented.	R	0
PS	18	Indicates that the paired-single (PS) floating-point data type and instructions are implemented: 0: PS floating-point not implemented 1: PS floating-point implemented This bit is always 0 to indicate that paired-single floating-point data types are not implemented.	R	0
D	17	Indicates that the double-precision (D) floating-point data type and instructions are implemented: 0: D floating-point not implemented 1: D floating-point implemented This bit is always 1 to indicate that double-precision floating-point data types are implemented.	R	1
S	16	Indicates that the single-precision (S) floating-point data type and instructions are implemented: 0: S floating-point not implemented 1: S floating-point implemented This bit is always 1 to indicate that single-precision floating-point data types are implemented.	R	1
Processor ID	15:8	Identifies the floating-point processor. This value matches the corresponding field of the CP0 PRId register.	R	0x81
Revision	7:0	Specifies the revision number of the FPU. This field allows software to distinguish between one revision and another of the same floating-point processor type. This value matches the corresponding field of the CP0 PRId register.	R	Hardwired
0	31:20	These bits must be written as zeros; they return zeros on reads.	0	0

3.5.2 Floating-Point Condition Codes Register (FCCR, CP1 Control Register 25)

The Floating-Point Condition Codes Register (*FCCR*) is an alternative way to read and write the floating-point condition code values that also appear in the *FCSR*. Unlike the *FCSR*, all eight FCC bits are contiguous in the *FCCR*. Figure 3-12 shows the format of the *FCCR*; Table 3-7 describes the *FCCR* bit fields.

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="24">0</td><td colspan="8">FCC</td></tr>
</table>

Figure 3-12 FCCR Format

Table 3-7 FCCR Bit Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
FCC	7:0	Floating-point condition code. Refer to the description of this field in Section 3.5.5, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
0	31:8	These bits must be written as zeros; they return zeros on reads.	0	0

3.5.3 Floating-Point Exceptions Register (FEXR, CP1 Control Register 26)

The Floating-Point Exceptions Register (*FEXR*) is an alternative way to read and write the Cause and Flags fields that also appear in the *FCSR*. Figure 3-13 shows the format of the *FEXR*; Table 3-8 describes the *FEXR* bit fields.

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="14">0</td><td colspan="6">Cause</td><td colspan="5">0</td><td colspan="5">Flags</td><td colspan="2">0</td></tr>
<tr><td colspan="14"></td><td>E</td><td>V</td><td>Z</td><td>O</td><td>U</td><td>I</td><td colspan="5"></td><td>V</td><td>Z</td><td>O</td><td>U</td><td>I</td><td colspan="2"></td></tr>
</table>

Figure 3-13 FEXR Format

Table 3-8 FEXR Bit Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
Cause	17:12	Cause bits. Refer to the description of this field in Section 3.5.5, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
Flags	6:2	Flag bits. Refer to the description of this field in Section 3.5.5, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
0	31:18, 11:7, 1:0	These bits must be written as zeros; they return zeros on reads.	0	0

3.5.4 Floating-Point Enables Register (FENR, CP1 Control Register 28)

The Floating-Point Enables Register (*FENR*) is an alternative way to read and write the Enables, FS, and RM fields that also appear in the *FCSR*. Figure 3-14 shows the format of the *FENR*; Table 3-9 describes the *FENR* bit fields.

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

Figure 3-14 FENR Format

Table 3-9 FENR Bit Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
Enables	11:7	Enable bits. Refer to the description of this field in Section 3.5.5, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
FS	2	Flush to Zero bit. Refer to the description of this field in Section 3.5.5, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
RM	1:0	Rounding mode. Refer to the description of this field in Section 3.5.5, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
0	31:12, 6:3	These bits must be written as zeros; they return zeros on reads.	0	0

3.5.5 Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)

The 32-bit Floating-Point Control and Status Register (*FCSR*) controls the operation of the FPU and shows the following status information:

- selects the default rounding mode for FPU arithmetic operations
- selectively enables traps of FPU exception conditions
- controls some denormalized number handling options
- reports any IEEE exceptions that arose during the most recently executed instruction
- reports any IEEE exceptions that cumulatively arose in completed instructions
- indicates the condition code result of FP compare instructions

Access to the *FCSR* is not privileged; it can be read or written by any program that has access to the FPU (via the coprocessor enables in the *Status* register). Figure 3-15 shows the format of the *FCSR*; Table 3-10 describes the *FCSR* bit fields.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCC							FS	FCC	FO	FN	0			Cause						Enables					Flags					RM	
7	6	5	4	3	2	1		0						E	V	Z	O	U	I	V	Z	O	U	I	V	Z	O	U	I		

Figure 3-15 FCSR Format

Table 3-10 FCSR Bit Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bit**			
FCC	31:25, 23	Floating-point condition codes. These bits record the result of floating-point compares and are tested for floating-point conditional branches and conditional moves. The FCC bit to use is specified in the compare, branch, or conditional move instruction. For backward compatibility with previous MIPS ISAs, the FCC bits are separated into two non-contiguous fields.	R/W	Undefined
FS	24	Flush to Zero (FS). Refer to Section 3.5.6, "Operation of the FS/FO/FN Bits" for more details on this bit.	R/W	Undefined
FO	22	Flush Override (FO). Refer to Section 3.5.6, "Operation of the FS/FO/FN Bits" for more details on this bit.	R/W	Undefined
FN	21	Flush to Nearest (FN). Refer to Section 3.5.6, "Operation of the FS/FO/FN Bits" for more details on this bit.	R/W	Undefined
Cause	17:12	Cause bits. These bits indicate the exception conditions that arise during execution of an FPU arithmetic instruction. A bit is set to 1 when the corresponding exception condition arises during the execution of an instruction; otherwise, it is cleared to 0. By reading the registers, the exception condition caused by the preceding FPU arithmetic instruction can be determined. Refer to Table 3-11 for the meaning of each cause bit.	R/W	Undefined
Enables	11:7	Enable bits. These bits control whether or not a trap is taken when an IEEE exception condition occurs for any of the five conditions. The trap occurs when both an enable bit and its corresponding cause bit are set either during an FPU arithmetic operation or by moving a value to the *FCSR* or one of its alternative representations. Note that Cause bit E (CauseE) has no corresponding enable bit; the MIPS architecture defines non-IEEE Unimplemented Operation exceptions as always enabled. Refer to Table 3-11 for the meaning of each enable bit.	R/W	Undefined
Flags	6:2	Flag bits. This field shows any exception conditions that have occurred for completed instructions since the flag was last reset by software. When an FPU arithmetic operation raises an IEEE exception condition that does not result in a Floating-Point Exception (the enable bit was off), the corresponding bit(s) in the Flags field are set, while the others remain unchanged. Arithmetic operations that result in a Floating-Point Exception (the enable bit was on) do not update the Flags field. Hardware never resets this field; software must explicitly reset this field. Refer to Table 3-11 for the meaning of each flag bit.	R/W	Undefined

Table 3-10 FCSR Bit Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bit			
RM	1:0	Rounding mode. This field indicates the rounding mode used for most floating-point operations (some operations use a specific rounding mode). Refer to Table 3-12 for the encoding of this field.	R/W	Undefined
0	20:18	These bits must be written as zeros; they return zeros on reads.	0	0

Table 3-11 Cause, Enables, and Flags Definitions

Bit Name	Bit Meaning
E	Unimplemented Operation (this bit exists only in the Cause field).
V	Invalid Operations
Z	Divide by Zero
O	Overflow
U	Underflow
I	Inexact

Table 3-12 Rounding Mode Definitions

RM Field Encoding	Meaning
0	RN - Round to Nearest Rounds the result to the nearest representable value. When two representable values are equally near, the result is rounded to the value whose least significant bit is zero (even).
1	RZ - Round Toward Zero Rounds the result to the value closest to but not greater in magnitude than the result.
2	RP - Round Towards Plus Infinity Rounds the result to the value closest to but not less than the result.
3	RM - Round Towards Minus Infinity Rounds the result to the value closest to but not greater than the result.

3.5.6 Operation of the FS/FO/FN Bits

The FS, FO, and FN bits in the CP1 *FCSR* register control handling of denormalized operands and *tiny* results (i.e. nonzero result between $\pm 2^{E\_min}$), whereby the FPU can handle these cases right away instead of relying on the much slower software handler. The trade-off is a loss of IEEE compliance and accuracy (except for use of the FO bit), because a minimal normalized or zero result is provided by the FPU instead of the more accurate denormalized result that a software handler would give. The benefit is a significantly improved performance and precision.

Use of the FS, FO, and FN bits affects handling of denormalized floating-point numbers and tiny results for the instructions listed below:

FS and FN bit:	ADD, CEIL, CVT, DIV, FLOOR, MADD, MSUB, MUL, NMADD, NMSUB, RECIP, ROUND, RSQRT, SQRT, TRUNC, SUB, ABS, C.cond, and NEG[a]
FO bit:	MADD, MSUB, NMADD, and NMSUB

[a] For ABS, C.cond, and NEG, denormal input operands or tiny results doe not result in Unimplemented exceptions when FS = 1. Flushing to zero nonetheless is implemented such that these operations return the same result as an equivalent sequence of arithmetic FPU operations.

Instructions not listed above do not cause Unimplemented Operation exceptions on denormalized numbers in operands or results.

Figure 3-16 depicts how the FS, FO, and FN bits control handling of denormalized numbers. For instructions that are not multiply or add types (such as DIV), only the FS and FN bits apply.

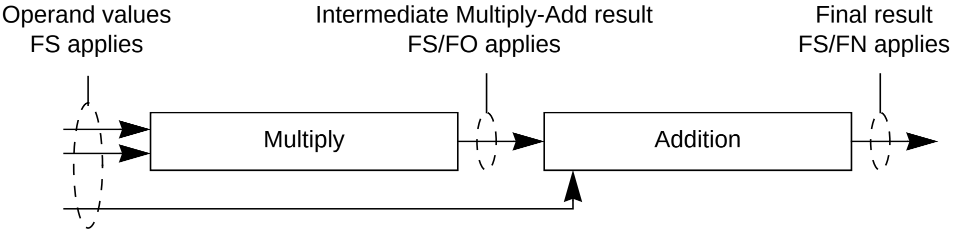


Figure 3-16 FS/FO/FN Bits Influence on Multiply and Addition Results

3.5.6.1 Flush To Zero Bit

When the Flush To Zero (FS) bit is set, denormal input operands are flushed to zero. Tiny results are flushed to either zero or the applied format's smallest normalized number (MinNorm) depending on the rounding mode settings. Table 3-13 lists the flushing behavior for tiny results..

Table 3-13 Zero Flushing for Tiny Results

Rounding Mode	Negative Tiny Result	Positive Tiny Result
RN (RM=0)	-0	+0
RZ(RM=1)	-0	+0
RP (RM=2)	-0	+MinNorm
RM (RM=3)	-MinNorm	+0

The flushing of results is based on an intermediate result computed by rounding the mantissa using an unbounded exponent range; that is, tiny numbers are not *normalized* into the supported exponent range by shifting in leading zeros prior to rounding.

Handling of denormalized operand values and tiny results depends on the FS bit setting as shown in Table 3-14.

Table 3-14 Handling of Denormalized Operand Values and Tiny Results Based on FS Bit Setting

FS Bit	Handling of Denormalized Operand Values
0	An Unimplemented Operation exception is taken.
1	Instead of causing an Unimplemented Operation exception, operands are flushed to zero, and tiny results are forced to zero or MinNorm.

3.5.6.2 Flush Override Bit

When the Flush Override (FO) bit is set, a tiny intermediate result of any multiply-add type instruction is not flushed according to the FS bit. The intermediate result is maintained in an internal normalized format to improve accuracy. FO only applies to the intermediate result of a multiply-add type instruction.

Handling of tiny intermediate results depends on the FO and FS bits as shown in Table 3-15.

Table 3-15 Handling of Tiny Intermediate Result Based on the FO and FS Bit Settings

FO Bit	FS Bit	Handling of Tiny Result Values
0	0	An Unimplemented Operation exception is taken.
0	1	The intermediate result is forced to the value that would have been delivered for an untrapped underflow (see Table 3-34) instead of causing an Unimplemented Operation exception.
1	Don't care	The intermediate result is kept in an internal format, which can be perceived as having the usual mantissa precision but with unlimited exponent precision and without forcing to a specific value or taking an exception.

3.5.6.3 Flush to Nearest

When the Flush to Nearest (FN) bit is set and the rounding mode is Round to Nearest (RN), a tiny final result is flushed to zero or MinNorm. If a tiny number is strictly below MinNorm/2, the result is flushed to zero; otherwise, it is flushed to MinNorm (see Figure 3-17). The flushed result has the same sign as the result prior to flushing. Note that the FN bit takes precedence over the FS bit.

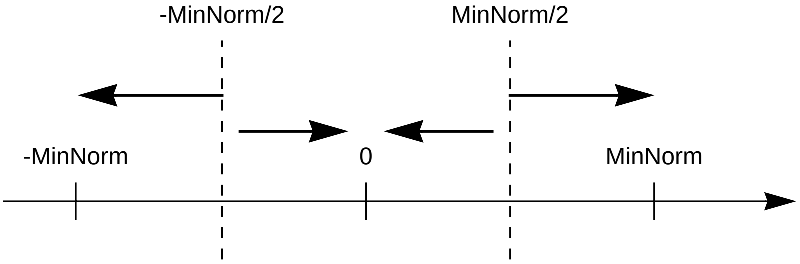


Figure 3-17 Flushing to Nearest when Rounding Mode is Round to Nearest

For all rounding modes other than Round to Nearest (RN), setting the FN bit causes final results to be flushed to zero or MinNorm as if the FS bit was set.

Handling of tiny final results depends on the FN and FS bits as shown in Table 3-16.

Table 3-16 Handling of Tiny Final Result Based on FN and FS Bit Settings

FN Bit	FS Bit	Handling of Tiny Result Values
0	0	An Unimplemented Operation exception is taken.
0	1	Final result is forced to the value that would have been delivered for an untrapped underflow (see Table 3-34) rather than causing an Unimplemented Operation exception.
1	Don't care	Final result is rounded to either zero or 2^{E_min} (MinNorm), whichever is closest when in Round to Nearest (RN) rounding mode. For other rounding modes, a final result is given as if FS was set to 1.

3.5.6.4 Recommended FS/FO/FN Settings

Table 3-17 summarizes the recommended FS/FO/FN settings.

Table 3-17 Recommended FS/FO/FN Settings

FS Bit	FO Bit	FN Bit	Remarks
0	0	0	IEEE-compliant mode. Low performance on denormal operands and tiny results.
1	0	0	Regular MIPS64 embedded applications. High performance on denormal operands and tiny results.
1	1	1	Highest accuracy and performance configuration.[a]

[a] Note that in this mode, MADD might return a different result other than the equivalent MUL and ADD operation sequence.

3.5.7 FCSR Cause Bit Update Flow

3.5.7.1 Exceptions Triggered by CTC1

Regardless of the targeted control register, the CTC1 instruction causes the Enables and Cause fields of the *FCSR* to be inspected in order to determine if an exception is to be thrown.

3.5.7.2 Generic Flow

Computations are performed in two steps:

1. Compute rounded mantissa with unbound exponent range.
2. Flush to default result if the result from Step #1 above is overflow or tiny (no flushing happens on denorms for instructions supporting denorm results, such as MOV).

The Cause field is updated after each of these two steps. Any enabled exceptions detected in these two steps cause a trap, and no further updates to the Cause field are done by subsequent steps.

Step #1 can set cause bits I, U, O, Z, V, and E. E has priority over V; V has priority over Z; and Z has priority over U and O. Thus when E, V, or Z is set in Step #1, no other cause bits can be set. However, note that I and V both can be set if a denormal operand was flushed (FS = 1). I, U, and O can be set alone or in pairs (IU or IO). U and O never can be set simultaneously in Step #1. U and O are set if the computed unbounded exponent is outside the exponent range supported by the normalized IEEE format.

Step #2 can set I if a default result is generated.

3.5.7.3 Multiply-Add Flow

For multiply-add type instructions, the computation is extended with two more steps:

1. Compute rounded mantissa with unbound exponent range for the multiply.
2. Flush to default result if the result from Step #1 is overflow or tiny (no flushing happens on tiny results if FO = 1).
3. Compute rounded mantissa with unbounded exponent range for the add.
4. Flush to default result if the result from Step #3 is overflow or tiny.

The Cause field is updated after each of these four steps. Any enabled exceptions detected in these four steps cause a trap, and no further updates to the Cause field are done by subsequent steps.

Step #1 and Step #3 can set a cause bit as described for Step #1 in Section 3.5.7.2, "Generic Flow".

Step #2 and Step #4 can set I if a default result is generated.

Although U and O can never both be set in Step #1 or Step #3, both U and O might be set after the multiply-add has executed in Step #3 because U might be set in Step #1 and O might be set in Step #3.

3.5.7.4 Cause Update Flow for Input Operands

Denormal input operands to Step #1 or Step #3 always set Cause bit I when FS = 1. For example, SNaN+DeNorm set I (and V) provided that Step #3 was reached (in case of a multiply-add type instruction).

Conditions directly related to the input operand (for example, I/E set due to DeNorm, V set due to SNaN and QNaN propagation) are detected in the step where the operand is logically used. For example, for multiply-add type instructions, exceptional conditions caused by the input operand fr are detected in Step #3.

3.5.7.5 Cause Update Flow for Unimplemented Operations

Note that Cause bit E is special; it clears any Cause updates done in previous steps. For example, if Step #3 caused E to be set, any I, U, or O Cause update done in Step #1 or Step #2 is cleared. Only E is set in the Cause field when an Unimplemented Operation trap is taken.

3.6 Instruction Overview

The functional groups into which the FPU instructions are divided are described in the following subsections:

- Section 3.6.1, "Data Transfer Instructions"
- Section 3.6.2, "Arithmetic Instructions"
- Section 3.6.3, "Conversion Instructions"
- Section 3.6.4, "Formatted Operand-Value Move Instructions"
- Section 3.6.5, "Conditional Branch Instructions"
- Section 3.6.6, "Miscellaneous Instructions"

The instructions are described in detail in Chapter 12, “Instructions,” on page 213, including descriptions of supported formats (fmt).

3.6.1 Data Transfer Instructions

The FPU has two separate register sets: coprocessor general registers (FPRs) and coprocessor control registers (FCRs). The FPU has a load/store architecture; all computations are done on data held in coprocessor general registers. The control registers are used to control FPU operation. Data is transferred between registers and the rest of the system with dedicated load, store, and move instructions. The transferred data is treated as unformatted binary data; no format conversions are performed, and therefore no IEEE floating-point exceptions can occur.

Table 3-18 lists the supported transfer operations.

Table 3-18 FPU Data Transfer Instructions

Transfer Direction			Data Transferred
FPU general register	↔	Memory	Word/doubleword load/store
FPU general register	↔	CPU general register	Word/doubleword move
FPU control register	↔	CPU general register	Word move

3.6.1.1 Data Alignment in Loads, Stores, and Moves

All coprocessor loads and stores operate on naturally aligned data items. An attempt to load or store to an address that is not naturally aligned for the data item causes an Address Error exception. Regardless of byte ordering (the endianness), the address of a word or doubleword is the smallest byte address in the object. For a big-endian machine, this is the most-significant byte; for a little-endian machine, this is the least-significant byte.

3.6.1.2 Addressing Used in Data Transfer Instructions

The FPU has loads and stores using the same register+offset addressing as that used by the CPU. Moreover, for the FPU only, there are load and store instructions using *register+register* addressing.

Tables 3-19 through 3-21 list the FPU data transfer instructions.

Table 3-19 FPU Loads and Stores Using Register+Offset Address Mode

Mnemonic	Instruction
LDC1	Load Doubleword to Floating Point
LWC1	Load Word to Floating Point
SDC1	Store Doubleword to Floating Point
SWC1	Store Word to Floating Point

Table 3-20 FPU Loads and Stores Using Register+Register Address Mode

Mnemonic	Instruction
LDXC1	Load Doubleword Indexed to Floating Point
LUXC1	Load Doubleword Indexed Unaligned to Floating Point
LWXC1	Load Word Indexed to Floating Point
SDXC1	Store Doubleword Indexed to Floating Point
SUXC1	Store Doubleword Indexed Unaligned to Floating Point
SWXC1	Store Word Indexed to Floating Point

Table 3-21 FPU Move To and From Instructions

Mnemonic	Instruction
CFC1	Move Control Word From Floating Point
CTC1	Move Control Word To Floating Point
DMFC1	Doubleword Move From Floating Point
DMTC1	Doubleword Move To Floating Point
MFC1	Move Word From Floating Point
MTC1	Move Word To Floating Point

3.6.2 Arithmetic Instructions

Arithmetic instructions operate on formatted data values. The results of most floating-point arithmetic operations meet IEEE Standard 754 for accuracy—a result is identical to an infinite-precision result that has been rounded to the specified format using the current rounding mode. The rounded result differs from the exact result by less than one Unit in the Least-significant Place (ULP).

In general, the arithmetic instructions take an Umimplemented Operation exception for denormalized numbers, except for the ABS, C, and NEG instructions, which can handle denormalized numbers. The FS, FO, and FN bits in the CP1 *FCSR* register can override this behavior as described in Section 3.5.6, "Operation of the FS/FO/FN Bits".

Table 3-22 lists the FPU IEEE compliant arithmetic operations.

Table 3-22 FPU IEEE Arithmetic Operations

Mnemonic	Instruction
ABS.fmt	Floating-Point Absolute Value
ADD.fmt	Floating-Point Add
C.cond.fmt	Floating-Point Compare
DIV.fmt	Floating-Point Divide
MUL.fmt	Floating-Point Multiply
NEG.fmt	Floating-Point Negate
SQRT.fmt	Floating-Point Square Root
SUB.fmt	Floating-Point Subtract

The two low latency operations, Reciprocal Approximation (RECIP) and Reciprocal Square Root Approximation (RSQRT), might be less accurate than the IEEE specification:

- The result of RECIP differs from the exact reciprocal by no more than one ULP.
- The result of RSQRT differs from the exact reciprocal square root by no more than two ULPs.

Table 3-23 lists the FPU-approximate arithmetic operations.

Table 3-23 FPU-Approximate Arithmetic Operations

Mnemonic	Instruction
RECIP.fmt	Floating-Point Reciprocal Approximation
RSQRT.fmt	Floating-Point Reciprocal Square Root Approximation

Four compound-operation instructions perform variations of multiply-accumulate operations; that is, multiply two operands, accumulate the result to a third operand, and produce a result. These instructions are listed in Table 3-24. The product is rounded according to the current rounding mode prior to the accumulation. This model meets the IEEE accuracy specification; the result is numerically identical to an equivalent computation using multiply, add, subtract, or negate instructions.

Table 3-24 FPU Multiply-Accumulate Arithmetic Operations

Mnemonic	Instruction
MADD.fmt	Floating-Point Multiply Add
MSUB.fmt	Floating-Point Multiply Subtract
NMADD.fmt	Floating-Point Negative Multiply Add
NMSUB.fmt	Floating-Point Negative Multiply Subtract

3.6.3 Conversion Instructions

These instructions perform conversions between floating-point and fixed-point data types. Each instruction converts values from a number of operand formats to a particular result format. Some conversion instructions use the rounding mode specified in the Floating Control/Status register (*FCSR*), while others specify the rounding mode directly.

Table 3-25 shows the supported operand range. An Unimplemented Operation exception is taken for convert instructions applied to numbers that fall outside of the corresponding range.

Table 3-25 Supported Operand Range for Convert Instructions

Operation	Supported Operand Range
Convert.S.W	0xFF800000 - 0x007FFFFF
Convert.S. L	0xFFFFFFFFFF800000 - 0x00000000007FFFFF
Convert.D.L	0xFFF8000000000000 - 0x0007FFFFFFFFFFFF
Convert.D.W	Can operate on full range
Convert.W.S	0xCAFFFFFF - 0x4AFFFFFF
Convert.W.D	0xC1CFFFFFFFFFFFFF - 0x41CFFFFFFFFFFFFF
Convert.L.S	0xCAFFFFFF - 0x4AFFFFFF
Convert.L.D	0xC32FFFFFFFFFFFFF - 0x432FFFFFFFFFFFFF
CVT.S.D	Can operate on full range[a]
CVT.D.S	Can operate on full range

[a] Large and small numbers can cause overflow respectively underflow.

In general, the conversion instructions take an Umimplemented Operation exception for denormalized numbers. The FS and FN bits in the CP1 *FCSR* register can override this behavior as described in Section 3.5.6, "Operation of the FS/FO/FN Bits".

Table 3-26 and Table 3-27 list the FPU conversion instructions according to their rounding mode.

Table 3-26 FPU Conversion Operations Using the FCSR Rounding Mode

Mnemonic	Instruction
CVT.D.fmt	Floating-Point Convert to Double Floating Point
CVT.L.fmt	Floating-Point Convert to Long Fixed Point
CVT.S.fmt	Floating-Point Convert to Single Floating Point
CVT.W.fmt	Floating-Point Convert to Word Fixed Point

Table 3-27 FPU Conversion Operations Using a Directed Rounding Mode

Mnemonic	Instruction
CEIL.L.fmt	Floating-Point Ceiling to Long Fixed Point
CEIL.W.fmt	Floating-Point Ceiling to Word Fixed Point
FLOOR.L.fmt	Floating-Point Floor to Long Fixed Point
FLOOR.W.fmt	Floating-Point Floor to Word Fixed Point
ROUND.L.fmt	Floating-Point Round to Long Fixed Point
ROUND.W.fmt	Floating-Point Round to Word Fixed Point
TRUNC.L.fmt	Floating-Point Truncate to Long Fixed Point
TRUNC.W.fmt	Floating-Point Truncate to Word Fixed Point

3.6.4 Formatted Operand-Value Move Instructions

These instructions move formatted operand values among FPU general registers. A particular operand type must be moved by the instruction that handles that type. There are three kinds of move instructions:

- Unconditional move
- Conditional move that tests an FPU true/false condition code
- Conditional move that tests a CPU general-purpose register against zero

Conditional move instructions operate in a way that might be unexpected. They always force the value in the destination register to become a value of the format specified in the instruction. If the destination register does not contain an operand of the specified format before the conditional move is executed, the contents become undefined. (For more information, see the individual descriptions of the conditional move instructions in the *MIPS64 Architecture Reference Manual, Volume II*.)

Table 3-28 through Table 3-30 list the formatted operand-value move instructions.

Table 3-28 FPU Formatted Operand Move Instruction

Mnemonic	Instruction
MOV.fmt	Floating-Point Move

Table 3-29 FPU Conditional Move on True/False Instructions

Mnemonic	Instruction
MOVF.fmt	Floating-Point Move Conditional on FP False
MOVT.fmt	Floating-Point Move Conditional on FP True

Table 3-30 FPU Conditional Move on Zero/Non-Zero Instructions

Mnemonic	Instruction
MOVN.fmt	Floating-Point Move Conditional on Nonzero
MOVZ.fmt	Floating-Point Move Conditional on Zero

3.6.5 Conditional Branch Instructions

The FPU has PC-relative conditional branch instructions that test condition codes set by FPU compare instructions (C.cond.fmt).

All branches have an architectural delay of one instruction. When a branch is taken, the instruction immediately following the branch instruction is said to be in the branch delay slot; it is executed before the branch to the target instruction takes place. Conditional branches come in two versions, depending upon how they handle an instruction in the delay slot when the branch is not taken and execution falls through:

- Branch instructions execute the instruction in the delay slot.
- Branch likely instructions do not execute the instruction in the delay slot if the branch is not taken (they are said to nullify the instruction in the delay slot).

  Although the Branch Likely instructions are included, software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

The MIPS64 architecture defines eight condition codes for use in compare and branch instructions. For backward compatibility with previous revisions of the ISA, condition code bit 0 and condition code bits 1 through 7 are in discontinuous fields in the *FCSR*.

Table 3-31 lists the conditional branch (branch and branch likely) FPU instructions; Table 3-32 lists the deprecated conditional branch likely instructions.

Table 3-31 FPU Conditional Branch Instructions

Mnemonic	Instruction
BC1F	Branch on FP False
BC1T	Branch on FP True

Table 3-32 Deprecated FPU Conditional Branch Likely Instructions

Mnemonic	Instruction
BC1FL	Branch on FP False Likely
BC1TL	Branch on FP True Likely

3.6.6 Miscellaneous Instructions

The MIPS64 architecture defines various miscellaneous instructions that conditionally move one CPU general register to another, based on an FPU condition code.

Table 3-33 lists these conditional move instructions.

Table 3-33 CPU Conditional Move on FPU True/False Instructions

Mnemonic	Instruction
MOVN	Move Conditional on FP False
MOVZ	Move Conditional on FP True

3.7 Exceptions

FPU exceptions are implemented in the MIPS FPU architecture with the Cause, Enables, and Flags fields of the *FCSR*. The flag bits implement IEEE exception status flags, and the cause and enable bits control exception trapping. Each field has a bit for each of the five IEEE exception conditions. The Cause field has an additional exception bit, Unimplemented Operation, used to trap for software emulation assistance. If an exception type is enabled through the Enables field of the *FCSR*, then the FPU is operating in precise exception mode for this type of exception.

3.7.1 Precise Exception Mode

In precise exception mode, a trap occurs before the instruction that causes the trap or any following instruction can complete and write its results. If desired, the software trap handler can resume execution of the interrupted instruction stream after handling the exception.

The Cause field reports per-bit instruction exception conditions. The cause bits are written during each floating-point arithmetic operation to show any exception conditions that arise during the operation. A cause bit is set to 1 if its corresponding exception condition arises; otherwise, it is cleared to 0.

A floating-point trap is generated any time both a cause bit and its corresponding enable bit are set. This case occurs either during the execution of a floating-point operation or when moving a value into the *FCSR*. There is no enable bit for Unimplemented Operations; this exception always generates a trap.

In a trap handler, exception conditions that arise during any trapped floating-point operations are reported in the Cause field. Before returning from a floating-point interrupt or exception, or before setting cause bits with a move to the *FCSR*, software first must clear the enabled cause bits by executing a move to the *FCSR* to prevent the trap from being erroneously retaken.

If a floating-point operation sets only non-enabled cause bits, no trap occurs and the default result defined by IEEE Standard 754 is stored (see Table 3-34). When a floating-point operation does not trap, the program can monitor the exception conditions by reading the Cause field.

The Flags field is a cumulative report of IEEE exception conditions that arise as instructions complete; instructions that trap do not update the flag bits. The flag bits are set to 1 if the corresponding IEEE exception is raised, otherwise the bits are unchanged. There is no flag bit for the MIPS Unimplemented Operation exception. The flag bits are never cleared as a side effect of floating-point operations, but they can be set or cleared by moving a new value into the *FCSR*.

3.7.2 Exception Conditions

The subsections below describe the following five exception conditions defined by IEEE Standard 754:

- Section 3.7.2.1, "Invalid Operation Exception"
- Section 3.7.2.2, "Division By Zero Exception"
- Section 3.7.2.3, "Underflow Exception"
- Section 3.7.2.4, "Overflow Exception"
- Section 3.7.2.5, "Inexact Exception"

Section 3.7.2.6, "Unimplemented Operation Exception" also describes a MIPS-specific exception condition, Unimplemented Operation Exception, that is used to signal a need for software emulation of an instruction. Normally an IEEE arithmetic operation can cause only one exception condition; the only case in which two exceptions can occur at the same time are Inexact With Overflow and Inexact With Underflow.

At the program’s direction, an IEEE exception condition can either cause a trap or not cause a trap. IEEE Standard 754 specifies the result to be delivered in case no trap is taken. The FPU supplies these results whenever the exception condition does not result in a trap. The default action taken depends on the type of exception condition and, in the case of the Overflow and Underflow, the current rounding mode. Table 3-34 summarizes the default results.

Table 3-34 Result for Exceptions Not Trapped

Bit	Description	Default Action
V	Invalid Operation	Supplies a quiet NaN.
Z	Divide by zero	Supplies a properly signed infinity.
U	Underflow	Depends on the rounding mode as shown below: 0 (RN) and 1 (RZ): Supplies a zero with the sign of the exact result. 2 (RP): For positive underflow values, supplies 2^{E_min} (MinNorm). For negative underflow values, supplies a positive zero. 3 (RM): For positive underflow values, supplies a negative zero. For negative underflow values, supplies a negative 2^{E_min} (MinNorm). Note that this behavior is only valid if the *FCSR* FN bit is cleared.
I	Inexact	Supplies a rounded result. If caused by an overflow without the overflow trap enabled, supplies the overflowed result. If caused by an underflow without the underflow trap enabled, supplies the overflowed result.
O	Overflow	Depends on the rounding mode, as shown below: 0 (RN): Supplies an infinity with the sign of the exact result. 1 (RZ): Supplies the format’s largest finite number with the sign of the exact result. 2 (RP): For positive overflow values, supplies positive infinity. For negative overflow values, supplies the format’s most negative finite number. 3 (RM): For positive overflow values, supplies the format’s largest finite number. For negative overflow values, supplies minus infinity.

3.7.2.1 Invalid Operation Exception

An Invalid Operation exception is signaled when one or both of the operands are invalid for the operation to be performed. When the exception condition occurs without a precise trap, the result is a quiet NaN.

The following operations are invalid:

- One or both operands are a signaling NaN (except for the non-arithmetic MOV.fmt, MOVT.fmt, MOVF.fmt, MOVN.fmt, and MOVZ.fmt instructions).
- Addition or subtraction: magnitude subtraction of infinities, such as (+∞) + (-∞) or (-∞) - (-∞).
- Multiplication: $0 \times \infty$, with any signs.
- Division: 0/0 or ∞/∞, with any signs.
- Square root: An operand of less than 0 (-0 is a valid operand value).
- Conversion of a floating-point number to a fixed-point format when either an overflow or an operand value of infinity or NaN precludes a faithful representation in that format.
- Some comparison operations in which one or both of the operands is a QNaN value.

3.7.2.2 Division By Zero Exception

The divide operation signals a Division By Zero exception if the divisor is zero and the dividend is a finite nonzero number. When no precise trap occurs, the result is a correctly signed infinity. Divisions (0/0 and ∞/0) do not cause the Division By Zero exception. The result of (0/0) is an Invalid Operation exception. The result of (∞/0) is a correctly signed infinity.

3.7.2.3 Underflow Exception

Two related events contribute to underflow:

- Tininess: The creation of a tiny, nonzero result between $\pm 2^{E\_min}$ which, because it is tiny, might cause some other exception later such as overflow on division. IEEE Standard 754 allows choices in detecting tininess events. The MIPS architecture specifies that tininess be detected after rounding, when a nonzero result computed as though the exponent range were unbounded would lie strictly between $\pm 2^{E\_min}$.
- Loss of accuracy: The extraordinary loss of accuracy occurs during the approximation of such tiny numbers by denormalized numbers. IEEE Standard 754 allows choices in detecting loss of accuracy events. The MIPS architecture specifies that loss of accuracy be detected as inexact result, when the delivered result differs from what would have been computed if both the exponent range and precision were unbounded.

The way that an underflow is signaled depends on whether or not underflow traps are enabled:

- When an underflow trap is not enabled, underflow is signaled only when both tininess and loss of accuracy have been detected. The delivered result might be zero, denormalized, or 2^{E_min}.
- When an underflow trap is enabled (through the *FCSR* Enables field), underflow is signaled when tininess is detected regardless of loss of accuracy.

3.7.2.4 Overflow Exception

An Overflow exception is signaled when the magnitude of a rounded floating-point result (if the exponent range is unbounded) is larger than the destination format’s largest finite number.

When no precise trap occurs, the result is determined by the rounding mode and the sign of the intermediate result.

3.7.2.5 Inexact Exception

An Inexact exception is signaled when one of the following occurs:

- The rounded result of an operation is not exact.
- The rounded result of an operation overflows without an overflow trap.
- When a denormal operand is flushed to zero.

3.7.2.6 Unimplemented Operation Exception

The Unimplemented Operation exception is a MIPS-defined exception that provides software emulation support. This exception is not IEEE-compliant.

The MIPS architecture is designed so that a combination of hardware and software can implement the architecture. Operations not fully supported in hardware cause an Unimplemented Operation exception, allowing software to perform the operation.

There is no enable bit for this condition; it always causes a trap. After the appropriate emulation or other operation is done in a software exception handler, the original instruction stream can be continued.

An Unimplemented Operation exception is taken in the following situations:

- when denormalized operands or tiny results are encountered for instructions not supporting denormal numbers and where such are not handed by the FS/FO/FN bits
- when a CVT instruction is applied with numbers out of the supported range.

3.8 Pipeline and Performance

This section describes the structure and operation of the FPU pipeline.

3.8.1 Pipeline Overview

The FPU has a seven stage pipeline to which the integer pipeline dispatches instructions. The FPU pipeline runs in parallel with the 5K integer pipeline. The FPU pipe is optimized for single-precision instructions, such that the basic multiply, ADD/SUB, and MADD/MSUB instructions can be performed with single-cycle throughput and low latency. Executing double-precision multiply and MADD/MSUB instructions requires a second pass through the M1 stage to generate all 64 bits of the product. Executing long latency instructions, such as DIV and RSQRT, extends the M1 stage. Figure 3-18 shows the FPU pipeline.

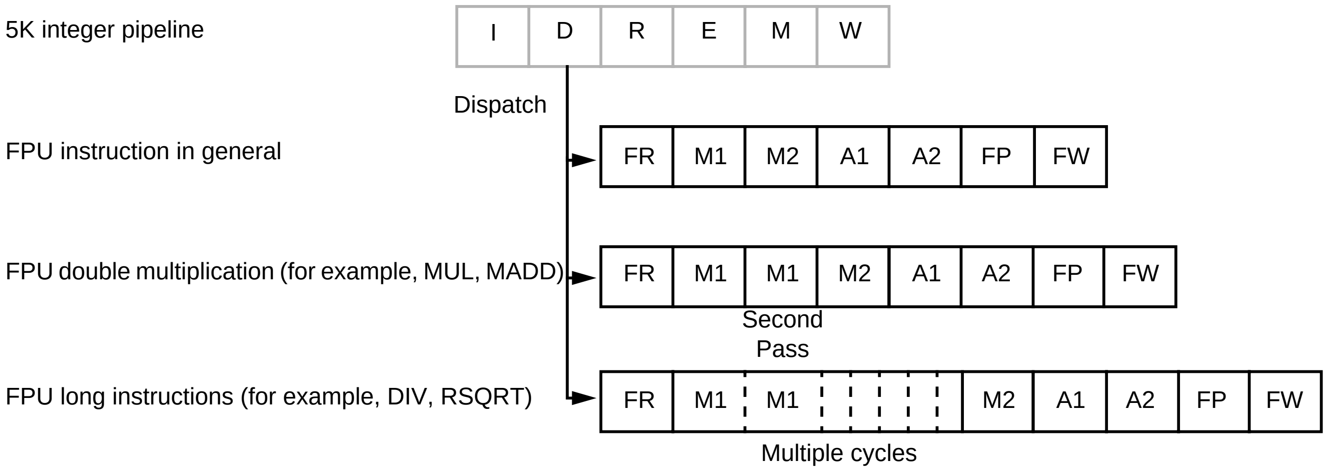

5K integer pipeline
I
D
R
E
M
W
Dispatch
FPU instruction in general
FR
M1
M2
A1
A2
FP
FW
FPU double multiplication (for example, MUL, MADD)
FR
M1
M1
M2
A1
A2
FP
FW
Second Pass
FPU long instructions (for example, DIV, RSQRT)
FR
M1
M1
M2
A1
A2
FP
FW
Multiple cycles

Figure 3-18 FPU Pipeline

3.8.1.1 FR Stage - Decode, Register Read, and Unpack

The FR stage has the following functionality:

- The dispatched instruction is decoded for register accesses.
- Data is read from the register file.
- The operands are unpacked into an internal format.

3.8.1.2 M1 Stage - Multiply Tree

The M1 stage has the following functionality:

- A single-cycle multiply array is provided for single-precision data format multiplication, and two cycles are provided for double-precision data format multiplication.
- The long instructions, such as divide and square root, iterate for several cycles in this stage.
- Sum of exponents is calculated.

3.8.1.3 M2 Stage - Multiply Complete

The M2 stage has the following functionality:

- Multiplication is complete when the carry-save encoded product is compressed into binary.
- Rounding is performed.
- Exponent difference for addition path is calculated.

3.8.1.4 A1 Stage - Addition First Step

This stage performs the first step of the addition.

3.8.1.5 A2 Stage - Addition Second and Final Step

This stage performs the second and final step of the addition.

3.8.1.6 FP Stage - Result Pack

The FP stage has the following functionality:

- The result coming from the datapath is packed into IEEE 754 Standard format for the FPR register file.
- Overflow and underflow exceptional conditions are resolved.

3.8.1.7 FW Stage - Register Write

The result is written to the FPR register file.

3.8.2 Bypassing

The FPU pipeline implements extensive bypassing, as shown in Figure 3-19. Results do not need to be written into the register file and read back before they can be used, but can be forwarded directly to an instruction already in the pipe. Some bypassing is disabled when operating in MIPS32 register file mode, the FP bit in the CP0 *Status* register is 0, due to the paired even-odd 32-bit registers that provide 64-bit registers.

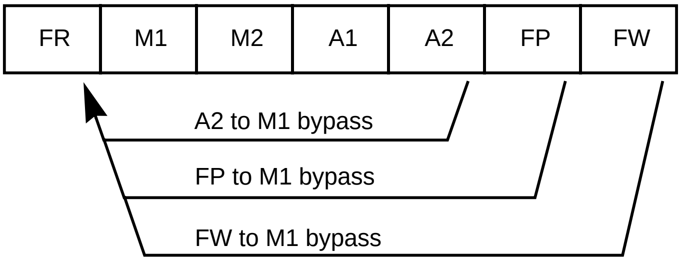


Figure 3-19 Arithmetic Pipeline Bypass Paths

3.8.3 Repeat Rate and Latency

Table 3-35 shows the repeat rate and latency for the FPU instructions.

Table 3-35 5Kf Core FPU Latency and Repeat Rate

Opcode[a]	Latency (cycles)	Repeat Rate (cycles)
ABS.[S,D], NEG.[S,D], ADD.[S,D], SUB.[S,D], MUL.S, MADD.S, MSUB.S, NMADD.S, NMSUB.S	4	1
MUL.D, MADD.D, MSUB.D, NMADD.D, NMSUB.D	5	2
RECIP.S	13	10
RECIP.D	25	21
RSQRT.S	17	14
RSQRT.D	35	31
DIV.S, SQRT.S	17	14
DIV.D, SQRT.D	32	29
C.cond.[S,D] to MOVF.fmt and MOVT.fmt instruction / MOVT, MOVN, BC1 instruction	1 / 2	1
CVT.D.S, CVT.[S,D].[W,L]	4	1

Table 3-35 5Kf Core FPU Latency and Repeat Rate (Continued)

Opcode[a]	Latency (cycles)	Repeat Rate (cycles)
CVT.S.D	6	1
CVT.[W,L].[S,D], CEIL.[W,L].[S,D], FLOOR.[W,L].[S,D], ROUND.[W,L].[S,D], TRUNC.[W,L].[S,D]	5	1
MOV.[S,D], MOVF.[S,D], MOVN.[S,D], MOVT.[S,D], MOVZ.[S,D]	4	1
LWC1, LDC1, LDXC1, LUXC1, LWXC1	3	1
MTC1, DMTC1, MFC1, DMFC1	2	1

[a] Format: S = Single, D = Double, W = Word, L = Longword.

Chapter 4

Memory Management

This chapter describes the 5K Memory Management Unit, including the Translation Lookaside Buffer and Fixed Mapping Translation option. It contains the following sections:

4.1 Introduction

The 5K microprocessor core assigns address translation and related virtual-memory support functions to a specialized processor called the Memory Management Unit (MMU). The MMU is incorporated in the CPU and is positioned between the CPU’s integer unit and main memory. The MMU receives virtual addresses from the integer unit and converts them to physical addresses for accessing the main memory or a cache system, if one is present. The MMU is also responsible for handling the control of cacheability and such exceptional conditions as memory protection violations.

The 5K core MMU can be implemented using either a translation lookaside buffer (TLB) or a simpler Fixed Mapping Translation (FMT) scheme. Using an FMT instead of the TLB function is supported only for the 32-bit addressing mode; a TLB supports both the 32- and 64-bit addressing modes. The FMT translates virtual addresses to physical addresses using a fixed-offset mechanism that depends on the current operating mode (User, Kernel, Supervisor, or Debug). The FMT option is described in the final section of this chapter, Section 4.8, "FMT Memory Maps"; the intervening discussion describes implementations using a TLB.

4.2 TLB Organization

The TLB is a fully-associative cache of virtual/physical address pairs that are used in the virtual-to-physical address translation. The 5K core supports implementations with a TLB containing 16, 32, or 48 dual entries. Each entry contains two logical components: a Tag comparison section (Tag/CAM[1] part) and a physical translation section (Data/RAM part). Figure 4-1 shows the logical arrangement of a TLB entry.

[1] CAM = Contents Addressable Memory

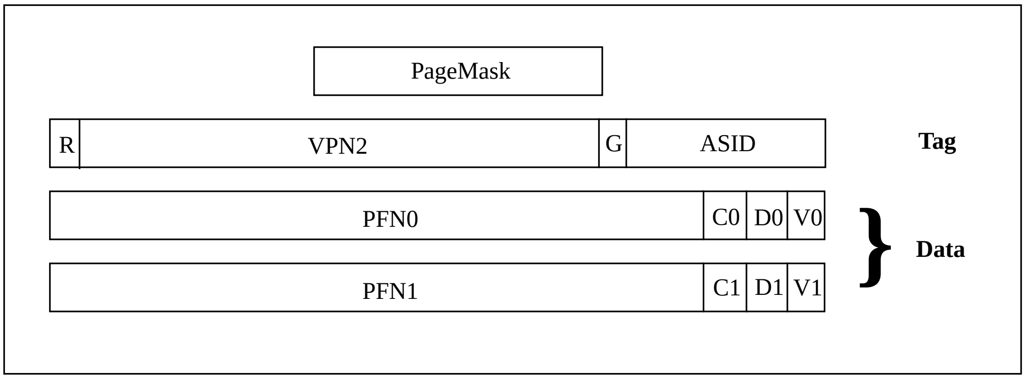


Figure 4-1 TLB Entry Format

The comparison section includes the mapping region specifier (R), the entry's Virtual Page Number VPN2 (VPN divided by 2), the ASID, the G(lobal) bit, and a PageMask field which allows different page sizes for all entries.

The physical translation section contains a pair of entries, each of which contains the physical page frame number (PFN), a valid (V) bit, a dirty (D) bit, and a cache coherency field (C).

Each VPN2 maps two consecutive VPNs to two independently specified physical pages, corresponding to the even and odd pages of the pair (PFN0 and PFN1). During translation, which of the two PFN entries is read is determined by the virtual address bit immediately to the right of the section masked with the PageMask entry (see Table 4-1 in Section 4.3, "Address Translation").

For purposes of managing TLB entries by both hardware and software, the fields of the TLB entry correspond exactly to the fields in the CP0 *PageMask*, *EntryHi*, *EntryLo0*, and *EntryLo1* registers. The *ASID* and *VPN2* are contained in the *EntryHi* register, the even page entries are contained in *EntryLo0*, and the odd page entries are in *EntryLo1*. These registers are described in detail in Chapter 6, “Coprocessor 0 Registers.”

4.2.1 PageMask Field

The 5K supports page sizes from 4Kbytes to 16 Mbytes, in multiples of 4Kbytes. The PageMask field contains a comparison mask that determines the page size for each TLB entry—only the unmasked bits of the corresponding VPN2 are used in the TLB comparison. Variable-size pages assist the operating system in controlling both the amount of mapped space and the replacement characteristics of various memory regions, including the ability to provide special-purpose maps. The CP0 *PageMask* register is loaded with the page size, which is then entered into the PageMask field when a new TLB entry is written.

4.2.2 ASID, GLOBAL, and R Bits

The 8-bit Address Space Identifier (ASID) is used by the operating system to uniquely identify the same virtual address across different processes, and thus is useful in reducing the frequency of TLB flushing during a context switch. The operating system assigns ASIDs to each process, and stores the ASID in the CP0 *EntryHi* register. During address translation, the ASID in the *EntryHi* register is compared with the ASID in the TLB entry. The Region (R) bits are used to select between the various address regions.

In certain cases, the operating system may wish to associate the same virtual address with all processes. To address this need, the TLB includes a Global (G) bit which, when set to one, overrides the result of the ASID comparison during translation.

4.2.3 Dirty Bit

The Dirty bit is used as a write-protect bit for a page. If this bit is set to one, the page can be written; if this bit is set to zero, any attempt to write to the page causes a TLB Modified exception (see Section 4.6, "TLB Exceptions").

This feature allows memory protection on a per-page basis, and is also used for paging by the operating system.

4.2.4 Cache/Coherency Attributes

Each page has a set of cache attributes associated with it. These attributes include information about cacheability and cache write policy. The following five configurations are permitted:

- Cacheable, Write-through, No Write-allocate
- Cacheable, Write-through, Write-allocate
- Uncached (Write-around)
- Cacheable, Write-back, Write-allocate
- Uncached accelerated (Write-around)

Cache coherency is not supported by the 5K core. All pages are non-coherent.

Cache attributes are explained in Chapter 8, “Cache Organization and Operation.”

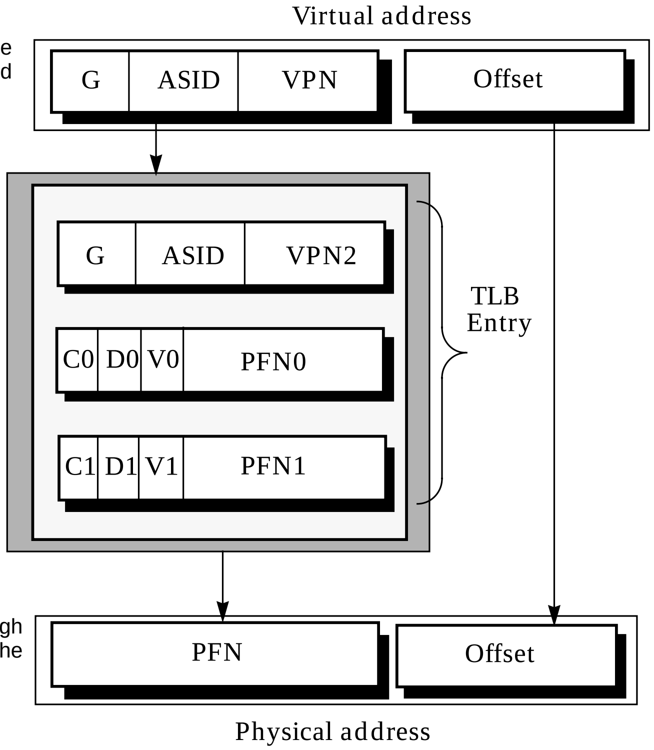
4.3 Address Translation

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 VPN of the address is the same as the VPN2 field of the entry, and either:

- The Global (G) bit of both the even and odd pages of the TLB entry is 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, we have a TLB *miss* and a refill exception is taken by the processor, and software can refill the TLB from a page table of virtual/physical addresses in memory.

Figure 4-2 shows a simplified view of virtual-to-physical address translation using the TLB.



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 (PFN0 or PFN1) 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.

Figure 4-2 Overview of Virtual-to-Physical Address Translation

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. The Offset does not pass through the TLB.

Figure 4-3 shows a more detailed view of address translation. The 5K core uses a 42-bit virtual address and a 36-bit physical address. The 42-bit virtual address is contained in VA[63:62] and VA[39:0].

The top portion of Figure 4-3 shows a virtual address for a 4-Kbyte page size. The width of the Offset is defined by the page size; the remaining 30 bits of the address represent the VPN used to index the 1Gbyte-entry page table.

The bottom portion of Figure 4-3 shows the virtual address for a 16-Mbyte page size (represented by a 24-bit Offset). The remaining 18 bits of the address represent the VPN, and index the 256Kbyte-entry page table.

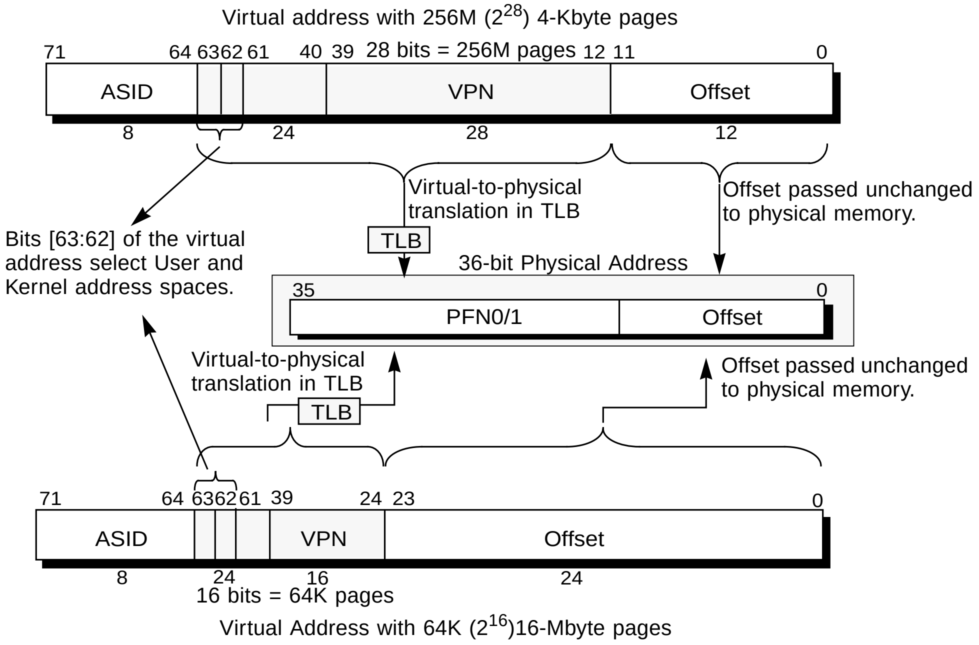

Virtual address with 256M (2^28) 4-Kbyte pages
71
64 63 62 61
40 39
28 bits = 256M pages
12 11
0
ASID
VPN
Offset
8
24
28
12
Virtual-to-physical translation in TLB
Offset passed unchanged to physical memory.
Bits [63:62] of the virtual address select User and Kernel address spaces.
TLB
36-bit Physical Address
35
0
PFN0/1
Offset
Virtual-to-physical translation in TLB
TLB
Offset passed unchanged to physical memory.
71
64 63 62 61
39
24 23
0
ASID
VPN
Offset
8
24
16
24
16 bits = 64K pages
Virtual Address with 64K (2^16)16-Mbyte pages

Figure 4-3 64-bit Virtual Address Translation

Table 4-1 shows the generation of the physical address as a function of the page size of the TLB entry matching the virtual address. The "Even/Odd Select" column indicates which virtual address bit is used to select between the even (*EntryLo0*) or odd (*EntryLo1*) entry in the matching TLB entry. The "PA Generated From" column specifies how the physical address is generated from the selected PFN and the Offset in the virtual address (the symbol || denotes "concatenation"). The PFN has the bit range $PFN_{23..0}$, corresponding to $PA_{35..12}$.

Table 4-1 Physical Address Generation

Page Size	Even/Odd Select	PA Generated From
4K Bytes	VA_{12}	$PFN_{23..0}$ || $VA_{11..0}$
16K Bytes	VA_{14}	$PFN_{23..2}$ || $VA_{13..0}$
64K Bytes	VA_{16}	$PFN_{23..4}$ || VA15..0
256K Bytes	VA_{18}	$PFN_{23..6}$ || $VA_{17..0}$
1M Bytes	VA_{20}	$PFN_{23..8}$ || $VA_{19..0}$
4M Bytes	VA_{22}	$PFN_{23..10}$ || $VA_{21..0}$
16M Bytes	VA_{24}	$PFN_{23..12}$ || $VA_{23..0}$

4.4 TLB Implementation Details

When configured with the TLB option, the 5K core MMU has in fact three TLBs: an instruction micro TLB (ITLB), a data micro TLB (DTLB), and a large joint TLB (JTLB). The purpose of the micro TLBs (uTLBs) is to increase the speed of translation and to allow two address translations to be performed simultaneously— one for an instruction fetch address (via the ITLB) and one for a data load/store address (via the DTLB).

The DTLB provides translations for data load/store instructions and operates as a fully-associative cache of the JTLB. Each data load/store instruction accesses the DTLB first. If a translation is not found in the DTLB, then the JTLB is accessed. Once the translation is retrieved, it is written back to the DTLB. Thus, the DTLB contains a subset of translations that are most-recently used. The same process occurs for instruction fetch addresses—the JTLB is accessed only when the instruction translation is not in the ITLB. A data load/store access has priority over an instruction fetch access when accessing the JTLB, because the load/store address belongs to an earlier instruction.

A ITLB/DTLB miss sequence (uTLB miss, JTLB lookup, uTLB update) has a penalty of two extra clock cycles. If we have simultaneous ITLB miss and DTLB miss, the DTLB gets first priority when accessing the JTLB, and I-access stalls an additional cycle, giving a total of 3 latency cycles.

The DTLB and ITLB each have four entries and map 4Kbyte pages only. Unlike the JTLB, they are managed by hardware and not accessible via software. Hardware guarantees that the micro TLBs are proper subsets of the JTLB, and that every translation in the uTLBs also exists in the JTLB.

uTLB refills use the least-recently-used (LRU) algorithm, in which the uTLB always replaces (refills) the entry which has not been accessed for the longest amount of time. Also, when one of the entries in the uTLB is flushed (invalidated) due to a JTLB write instruction, that entry becomes the LRU entry.

The mechanism used by hardware for implementing a partial flush of the uTLB ensures that whenever a TLB write instruction is executed, only the uTLB entries with matching index are invalidated. This effectively guarantees that the contents of the uTLB are always a valid subset of the JTLB.

4.5 TLB Management Instructions

The TLB management instructions are used to read, write, and probe entries in the TLB. These instructions are described in the following subsections.

4.5.1 TLBWI - TLB Write Indexed

A TLBWI instruction is used to refill a TLB entry, using an index contained in the CP0 *Index* register. Data in the CP0 *EntryHi* and *PageMask* registers is written into the TLB Tag entry, and data from CP0 *EntryLo0/1* is written into the TLB Data entry.

The TLBWI instruction executes in four cycles. In the first cycle, a write comparison is performed, in which the entry to be written is checked against existing TLB entries to determine whether the new virtual address is already resident in the TLB (at a different entry than the one to be written). The functionality of this write comparison differs from a normal compare in the following ways:

- A hardware reset bit (I) is implemented in the Tag entry to prevent power-up tag values from causing false write-compare matches.
- The incoming PageMask value must qualify the per-entry address comparison. (For example, if a 16Mbyte page is to be written and a 4Kbyte page within the 16Mbyte page is already in the TLB, then the write comparison must match for this case.)
- The incoming Global bit must be ORed with the per-entry Global bit, since an incoming entry with global enable must compare to an existing entry with global disable.
- The entry to be written is excluded from the compare, since overwrites of the same entry are permitted.

If the write comparison results in a match, the TLB Shutdown (TS) bit is set in the CP0 *Status* register, and a Machine Check exception is taken (refer to Section 4.6, "TLB Exceptions"). In this case, the TLB write is not performed and the entry remains unchanged.

Note that it is possible for multiple entries to match in the TLB during the write compare. For example, consider the case where 4Kbyte pages are resident in the TLB, and a write is attempted with a 16Mbyte page that includes all the resident 4Kbyte pages. There will be a match on all of the 4Kbyte entries *except* the entry to be written, since it is excluded. Thus, depending on the implementation, the match-line selection of PFN0/1 wordlines may need to be suppressed during the write compare cycle, since more than one could result in a match.

The second cycle of the TLBWI is only present because of timing considerations. In the third cycle of the TLBWI instruction, the TLB Tag entry and the *PFN0* of the Data entry are written. In the fourth and final cycle, the *PFN1* of the TLB Data entry is written.

4.5.2 TLBWR - TLB Write Random

The TLBWR instruction is similar to the TLBWI instruction except that the index used to access the TLB entry is contained in the CP0 *Random* register instead of the CP0 *Index* register. The index for the TLBWR is pseudo-randomly generated.

4.5.3 TLBP - TLB Probe

The TLB probe instruction is used to check for a specific VPN in the TLB. It performs a compare of VPN in the CP0 *EntryHi* register against all Tags in the TLB CAM. If there is a match, the index of the matching entry and the probe hit/miss indicator is written to the CP0 *Index* register. If there is no match, the value of the CP0 *Index* register is unpredictable, but the probe hit/miss indicator is still written to the CP0 *Index* register. Thus the CP0 *Index* register is always updated on a TLB Probe Operation, whereas the CP0 *EntryHi* and *EntryLo* registers are unchanged.

A TLBP instruction executes in one CPU clock cycle.

4.5.4 TLBR - TLB Read Indexed

The TLBR instruction is used to read a specific entry in the TLB pointed to by the CP0 *Index* register. Data from the indexed TLB Tag part (VPN and PageMask) is stored in the CP0 *EntryHi* and *PageMask* registers, and data from the indexed TLB Data part is stored in the CP0 *EntryLo0/1* registers.

A TLBR instruction executes in two CPU clock cycles.

4.6 TLB Exceptions

TLB exception conditions are listed and briefly described in Table 4-2. Exceptions are listed in the order of their relative priority, from high to low. A more detailed description of TLB exceptions is provided in Chapter 5, “Exception Processing.”

Table 4-2 TLB Exceptions

Exception	Description
TLB Refill - Instruction fetch	TLB miss occurred on an instruction fetch.
TLB Invalid - Instruction fetch	The valid bit was zero in the TLB entry matching the address referenced by an instruction fetch.
TLB Refill - Data access	TLB miss occurred on a data access.
TLB Invalid - Data access	The valid bit was zero in the TLB entry matching the address referenced by a load or store instruction.

Table 4-2 TLB Exceptions

Exception	Description
TLB Modified - Data access	The dirty bit was zero in the TLB entry matching the address referenced by a store instruction.

4.6.1 TLB Refill Exception

A TLB Refill or XTLB (extended TLB) Refill exception occurs when no TLB entry matches a reference to a mapped address space, and the *EXL* bit in the CP0 *Status* register is zero. Refer to Section 5.7, "TLB and XTLB Refill Exceptions" on page 87.

The occurrence of the TLB Refill exception during address translation is shown in Figure 4-4.

4.6.2 TLB Invalid Exception

A TLB invalid exception occurs when a TLB entry matches a reference to a mapped address space, but the matched entry has the Valid bit set to zero. Refer to Section 5.8, "TLB Invalid Exception" on page 88.

The occurrence of the TLB Invalid exception during address translation is shown in Figure 4-4.

4.6.3 TLB Modified Exception

A TLB Modified exception occurs when a store instruction references a mapped address, and the matching TLB entry is write-protected (that is, the entry's Dirty bit is zero, indicating that the entry cannot be modified). Refer to Section 5.9, "TLB Modified Exception" on page 89.

The occurrence of the TLB Modified exception during address translation is shown in Figure 4-4 below.

4.6.4 Machine Check (TLB Shutdown)

A Machine Check exception occurs when there is an attempt to execute a TLB write instruction that would result in multiple matching entries in the TLB.

The Machine Check exception is not shown in Figure 4-4, since it does not occur during the translation process.

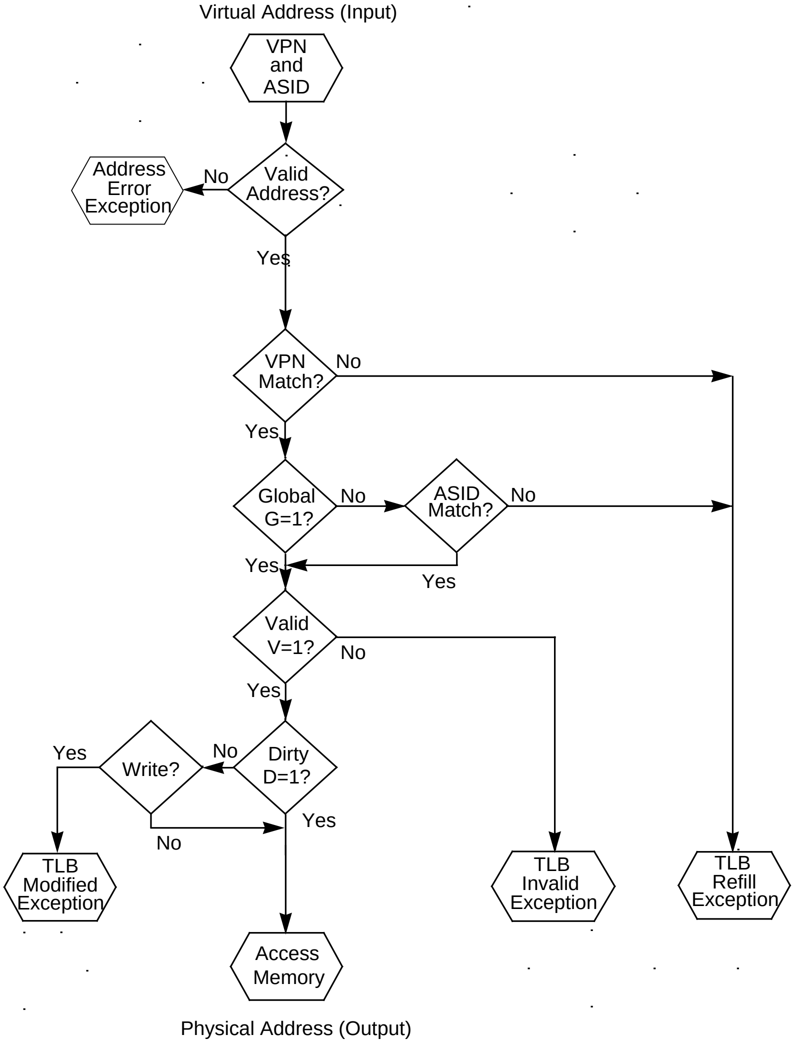


Figure 4-4 TLB Address Translation and Exception Conditions

4.7 TLB Memory Maps

With support for 64-bit operations and address calculation, the MIPS64 architecture implicitly defines and provides support for a 64-bit virtual address space, subdivided into four segments selected by bits 63:62 of the virtual address. To provide compatibility for 32-bit programs and MIPS32 processors, a 2^{32}-byte Compatibility address space is defined, separated into two non-contiguous ranges in which the upper 32 bits of the 64-bit address are the sign extension of bit 31. The Compatibility address space is similarly sub-divided into segments selected by bits 31:29 of the virtual address.

Each segment of an address space is classified as *mapped* or *unmapped*. A mapped address is one that is translated using the TLB or other translation unit. An unmapped address is one which is not translated and which provides a window into the lowest portion of the physical address space, starting at physical address zero, and with a size corresponding to the size of the unmapped segment.

Additionally, the kseg1 segment is classified as *uncached*. References to this segment bypass the cache hierarchy and allow direct access to memory.

Figure 4-5 shows the basic layout of the address spaces, including the Compatibility address space and the segments in each address space. Table 4-3 describes the address spaces in greater detail.

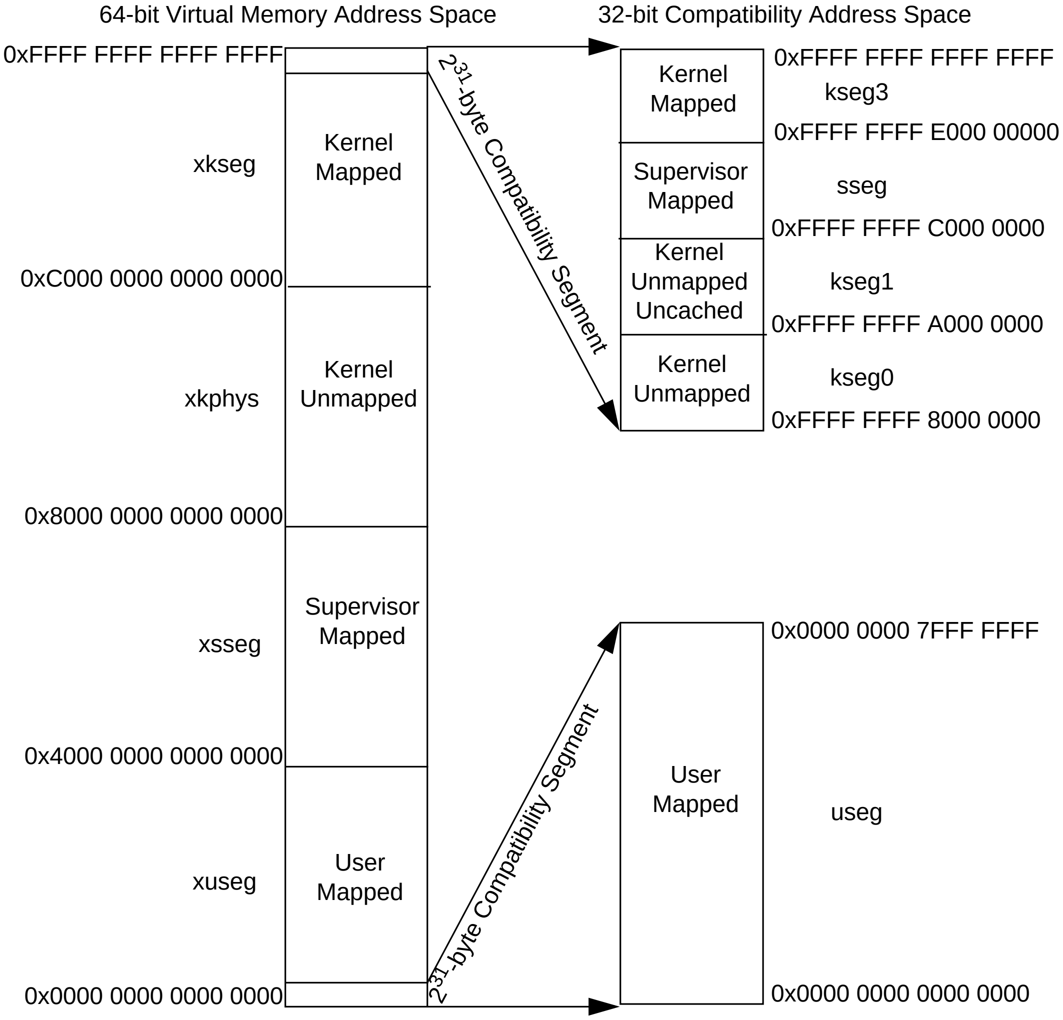


Figure 4-5 Virtual Address Spaces

Table 4-3 Virtual Address Spaces

$VA_{63..62}$	Segment Name(s)	Address Range	64-bit Enable	Associated with Mode	Reference Legal From Mode(s)	Segment Size	Segment Type
11_2	kseg3	0xFFFF FFFF FFFF FFFF through 0xFFFF FFFF E000 0000	Always	Kernel	Kernel	2^{29} bytes	32-bit Compatibility
	sseg, ksseg	0xFFFF FFFF DFFF FFFF through 0xFFFF FFFF C000 0000	Always	Supervisor	Supervisor, Kernel	2^{29} bytes	32-bit Compatibility
	kseg1	0xFFFF FFFF BFFF FFFF through 0xFFFF FFFF A000 0000	Always	Kernel	Kernel	2^{29} bytes	32-bit Compatibility
	kseg0	0xFFFF FFFF 9FFF FFFF through 0xFFFF FFFF 8000 0000	Always	Kernel	Kernel	2^{29} bytes	32-bit Compatibility
	xkseg	0xFFFF FFFF 7FFF FFFF through 0xC000 0000 0000 0000	KX	Kernel	Kernel	$(2^{40} - 2^{31})$ bytes	64-bit
10_2	xkphys	0xBFFF FFFF FFFF FFFF through 0x8000 0000 0000 0000	KX	Kernel	Kernel	8 2^{36}-byte regions within the 2^{62} byte segment	64-bit
01_2	xksseg	0x7FFF FFFF FFFF FFFF through 0x4000 0000 0000 0000	SX	Supervisor	Supervisor, Kernel	2^{40} bytes	64-bit
00_2	xuseg xkuseg	0x3FFF FFFF FFFF FFFF through 0x0000 0000 8000 0000	UX	User	User Supervisor Kernel	2^{40} bytes	64-bit
	useg kuseg	0x0000 0000 7FFF FFFF through 0x0000 0000 0000 0000	Always	User	User Supervisor Kernel	2^{31} bytes	32-bit Compatibility

The 5K core implements all four operating modes: User, Kernel, Supervisor, and Debug Modes.

Each segment of an address space is associated with one of the operating modes. A segment that is associated with a particular mode is accessible if the processor is running in that mode or in a more privileged mode—for example, a segment associated with User Mode is accessible when the processor is running in User or Kernel Modes. A segment is not accessible if the processor is running in a less privileged mode than the mode associated with the segment—for example, a segment associated with Kernel Mode is not accessible when the processor is running in User Mode, and such a reference results in an Address Error exception. The “Reference Legal from Mode(s)” column in Table 4-3 lists the modes from which each segment may be legally referenced.

If a segment has more than one name, each name denotes the mode from which the segment is referenced. For example, the segment name “useg” denotes a reference from User Mode, while the segment name “kuseg” denotes a reference to the same segment from Kernel Mode.

References to 64-bit segments (as shown in the “Segment Type” column of Table 4-3) are enabled only if the appropriate 64-bit Address Enable bit (*PX*, *UX*, or *KX*) is set. References to 32-bit Compatibility segments are always enabled.

4.7.1 Access Control as a Function of Address and Operating Mode

Table 4-4 describes the action taken by the processor for each section of the 64-bit address space as a function of the operating mode of the processor, including the selection of a TLB Refill vector and other special-case behavior.

Table 4-4 Address Space Access and TLB Refill Selection as a Function of Operating Mode

Virtual Address Range	Segment Name(s)	Action when Referenced from Operating Modes		
SEGBITS = 40, *PABITS* = 36		User Mode	Supervisor Mode	Kernel Mode
0xFFFF FFFF FFFF FFFF through 0xFFFF FFFF E000 0000	kseg3	Address Error	Address Error	Mapped Refill Vector: TLB (KX=0) XTLB(KX=1) See Section 4.7.5, "Address Translation in Debug Mode" on page 75 for special behavior when Debug_{DM} = 1
0xFFFF FFFF DFFF FFFF through 0xFFFF FFFF C000 0000	sseg, ksseg	Address Error	Mapped Refill Vector: TLB (KX=0) XTLB(KX=1)	Mapped Refill Vector: TLB (KX=0) XTLB(KX=1)
0xFFFF FFFF BFFF FFFF through 0xFFFF FFFF A000 0000	kseg1	Address Error	Address Error	Unmapped, Uncached See Section 4.7.2, "Address Translation and Cache Coherency Attributes for kseg0 and kseg1" on page 72
0xFFFF FFFF 9FFF FFFF through 0xFFFF FFFF 8000 0000	kseg0	Address Error	Address Error	Unmapped See Section 4.7.2, "Address Translation and Cache Coherency Attributes for kseg0 and kseg1" on page 72
0xFFFF FFFF 7FFF FFFF through 0xC000 00FF 8000 0000		Address Error	Address Error	Address Error
0xC000 00FF 7FFF FFFF through 0xC000 0000 0000 0000	xkseg	Address Error	Address Error	Address Error if KX = 0 Mapped if KX = 1 Refill Vector: XTLB

Table 4-4 Address Space Access and TLB Refill Selection as a Function of Operating Mode (Continued)

Virtual Address Range	Segment Name(s)	Action when Referenced from Operating Modes		
***SEGBITS* = 40, *PABITS* = 36**		**User Mode**	**Supervisor Mode**	**Kernel Mode**
0xBFFF FFFF FFFF FFFF through 0x8000 0000 0000 0000	xkphys	Address Error	Address Error	Address Error if KX = 0 or in certain address ranges within the segment Unmapped See Section 4.7.3, "Address Translation and Cache Coherency Attributes for xkphys" on page 72
0x7FFF FFFF FFFF FFFF through 0x4000 0100 0000 0000		Address Error	Address Error	Address Error
0x4000 00FF FFFF FFFF through 0x4000 0000 0000 0000	xsseg xksseg	Address Error	Address Error if SX = 0 Mapped if SX = 1 Refill Vector: XTLB	Address Error if SX = 0 Mapped if SX = 1 Refill Vector: XTLB
0x3FFF FFFF FFFF FFFF through 0x0000 0100 0000 0000		Address Error	Address Error	Address Error
0x0000 00FF FFFF FFFF through 0x0000 0000 8000 0000	xuseg xsuseg xkuseg	Address Error if UX = 0 Mapped if UX = 1 Refill Vector: XTLB	Address Error if UX = 0 Mapped if UX = 1 Refill Vector: XTLB	Address Error if UX = 0 Mapped if UX = 1 Refill Vector: XTLB See Section 4.7.4, "Address Translation for kuseg when StatusERL = 1" on page 74 for implementation-dependent behavior when Status$_{ERL}$=1

Table 4-4 Address Space Access and TLB Refill Selection as a Function of Operating Mode (Continued)

Virtual Address Range	**Segment Name(s)**	**Action when Referenced from Operating Modes**		
***SEGBITS* = 40, *PABITS* = 36**		**User Mode**	**Supervisor Mode**	**Kernel Mode**
0x0000 0000 7FFF FFFF through 0x0000 0000 0000 0000	useg suseg kuseg	Mapped Refill Vector: TLB (UX=0) XTLB(UX=1)	Mapped Refill Vector: TLB (UX=0) XTLB(UX=1)	Unmapped if $\text{Status}_{\text{ERL}}$=1 See Section 4.7.4, "Address Translation for kuseg when StatusERL = 1" on page 74 Mapped if $\text{Status}_{\text{ERL}}$=0 Refill Vector: TLB (UX=0) XTLB(UX=1)

4.7.2 Address Translation and Cache Coherency Attributes for kseg0 and kseg1

The kseg0 and kseg1 unmapped segments provide a window into the least-significant 2^{29} bytes of physical memory, and, as such, are not translated using the TLB or other address translation unit. The cache coherency attribute of kseg0 is supplied by the *K0* field of the *Config* register. The cache coherency attribute for the kseg1 segment is always uncached. Table 4-5 describes how this transformation is done, and the source of the cache coherency attributes for each segment.

Table 4-5 Address Translation and Cache Attributes for kseg0 and kseg1

Segment Name	**Virtual Address Range**	**Generates Physical Address**	**Cache Attribute**
kseg1	0xFFFF FFFF BFFF FFFF through 0xFFFF FFFF A000 0000	0x0000 0000 1FFF FFFF through 0x0000 0000 0000 0000	Uncached
kseg0	0xFFFF FFFF 9FFF FFFF through 0xFFFF FFFF 8000 0000	0x0000 0000 1FFF FFFF through 0x0000 0000 0000 0000	From K0 field of *Config* Register

4.7.3 Address Translation and Cache Coherency Attributes for xkphys

The xkphys is an unmapped segment composed of 8 address ranges, each of which provides a window into the entire 2^{PABITS} bytes of physical memory. For this segment, the cache coherency attribute is specified in $VA_{61..59}$ and has the same encoding as that shown in Table 6-6 on page 107. An Address Error exception occurs if $VA_{58..PABITS}$ are non-zero. If no Address Error exception occurs, the physical address is taken from $VA_{PABITS\text{-}1..0}$. Figure 4-6 shows the interpretation of the various fields of the virtual address when referencing the xkphys segment.

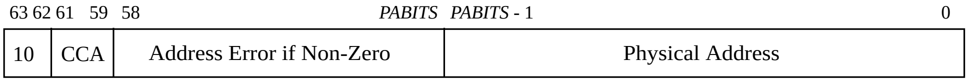


Figure 4-6 Address Interpretation for xkphys Segment

Table 4-6 Address Translation and Cache Attributes for xkphys

Virtual Address Range	Generates Physical Address	Cache Attribute
Assuming *PABITS* = 36		
0xBFFF FFFF FFFF FFFF through 0xB800 0010 0000 0000	Address Error	N/A
0xB800 000F FFFF FFFF through 0xB800 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Uses encoding 7 of Table 6-6 on page 107
0xB7FF FFFF FFFF FFFF through 0xB000 0010 0000 0000	Address Error	N/A
0xB000 000F FFFF FFFF through 0xB000 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Uses encoding 6 of Table 6-6 on page 107
0xAFFF FFFF FFFF FFFF through 0xA800 0010 0000 0000	Address Error	N/A
0xA800 000F FFFF FFFF through 0xA800 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Uses encoding 5 of Table 6-6 on page 107
0xA7FF FFFF FFFF FFFF through 0xA000 0010 0000 0000	Address Error	N/A
0xA000 000F FFFF FFFF through 0xA000 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Uses encoding 4 of Table 6-6 on page 107
0x9FFF FFFF FFFF FFFF through 0x9800 0010 0000 0000	Address Error	N/A

Table 4-6 Address Translation and Cache Attributes for xkphys (Continued)

Virtual Address Range Assuming *PABITS* = 36	Generates Physical Address	Cache Attribute
0x9800 000F FFFF FFFF through 0x9800 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Cacheable (see encoding 3 of Table 6-6 on page 107
0x97FF FFFF FFFF FFFF through 0x9000 0010 0000 0000	Address Error	N/A
0x9000 000F FFFF FFFF through 0x9000 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Uncached (see encoding 2 of Table 6-6 on page 107
0x8FFF FFFF FFFF FFFF through 0x8800 0010 0000 0000	Address Error	N/A
0x8800 000F FFFF FFFF through 0x8800 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Uses encoding 1 of Table 6-6 on page 107
0x87FF FFFF FFFF FFFF through 0x8000 0010 0000 0000	Address Error	N/A
0x8000 000F FFFF FFFF through 0x8000 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Uses encoding 0 of Table 6-6 on page 107

4.7.4 Address Translation for kuseg when Status$_{ERL}$ = 1

To provide support for the cache error exception handler, kuseg becomes an unmapped, uncached segment, similar to kseg1, when the *Status* register's *ERL* bit is set. This allows the cache error exception code to operate uncached using GPR R0 as a base register in order to save other GPRs before they are used by the exception handler.

The 5K core transforms all the lower 2^{36} bytes of kuseg.

4.7.5 Address Translation in Debug Mode

In Debug Mode, the EJTAG block treats the virtual address range 0xFFFF FFFF FF20 0000 through 0xFFFF FFFF FF3F FFFF, inclusive, as a special memory-mapped region referred to as the *dseg* region. The dseg address region is described in Table 4-7.

Table 4-7 Physical Address and Cache Attribute for dseg

Segment Name	Virtual Address	Generates Physical Address	Cache Attribute
dseg	0xFFFF.FFFF.FF20.0000 through 0xFFFF.FFFF.FF3F.FFFF	References are NOT mapped through the TLB. The physical address is obtained by stripping a number of the MSBs from the virtual address. Transactions to dseg do not occur on the external system memory interface.	Uncached

CPU access to the dseg address range is determined by the state of the Load Store Normal Memory (*LSNM*) bit in the *Debug* register, as shown in Table 4-8.

Table 4-8 CPU Access to dseg Address Range

Transaction	LSNM bit in Debug register	Access
Fetch	x	dseg address space
Load/Store	1	Kernel Mode address space
Load/Store	0	dseg address space
'x' denotes don't care.		

In Debug Mode, accesses outside the dseg address range are handled in the same way as accesses in non-Debug Mode.

Refer to Chapter 10, "EJTAG Debug Features," for a detailed description of Debug Mode.

4.8 FMT Memory Maps

Use of a FMT for the TLB function is only supported for the 32-bit addressing mode (the *Status* register's *KX*, *SX* and *UX* bits are set to zero). When the 5K core is configured with the FMT option, $Status_{KX,\ SX\ and\ UX}$ and the MMU Size field in the *Config1* register are hardwired to 0, and the *MT* field in the *Config* Register is set to the value 3.

The FMT translates virtual addresses to physical addresses using a fixed-offset mechanism that depends on the current operating mode (User or Kernel) and the value of the Error Level bit (*ERL*) in the CP0 *Status* register.

When using an FMT, cacheability of addresses is controlled by the *K0*, *KU*, and *K23* fields in the CP0 *Config* register. Refer to the description of the *Config* register in Chapter 6, "Coprocessor 0 Registers." Note that there is no address protection mechanism in the FMT implementation.

4.8.1 User Mode (useg/suseg/kuseg)

In User Mode, the physical address (PA) is determined by adding a fixed offset (0x4000 0000) to the virtual address (VA) when *ERL* equals zero. Thus VA=(0x0000 0000 to 0x7FFF FFFF) maps to PA=(0x4000 0000 to 0xBFFF FFFF). Refer to Figure 4-7.

When *ERL* is set, the physical address is equal to the virtual address. Refer to Figure 4-8.

4.8.2 Supervisor Mode (sseg)

Addresses in sseg are unmapped (VA=PA) and not affected by the *ERL* bit. Refer to Figure 4-7 and Figure 4-8.

4.8.3 Kernel Mode (kseg0, kseg1 and kseg3)

For kseg0 and kseg1, the translation used is identical to that used for TLB implementations, namely, both VA=0x8000 0000 to 0x9FFF FFFF (kseg0) and VA=0xA000 0000 to 0xBFFF FFFF (kseg1) map to PA=0x0000 0000 to 0x1FFF FFFF. Mapping is independent of the *ERL* bit. Refer to Figure 4-7 and Figure 4-8.

Addresses in kseg3 are unmapped (VA=PA) and not affected by the *ERL* bit. Refer to Figure 4-7 and Figure 4-8.

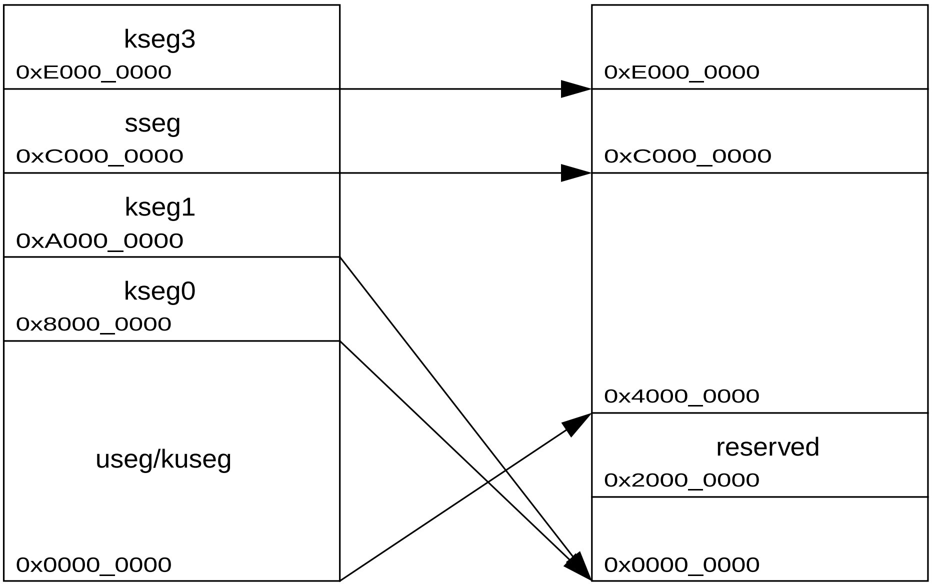


Figure 4-7 FMT Memory Map (ERL=0)

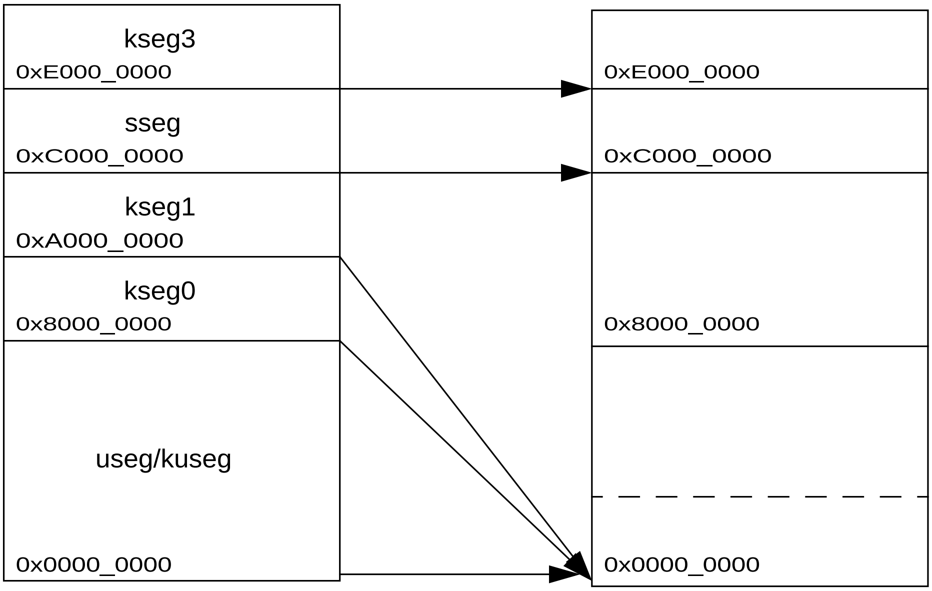


Figure 4-8 FMT Memory Map (ERL=1)

4.8.4 Debug Mode

In Debug Mode with the FMT option, address mapping is the same as the mapping used for debug-mode addresses with a TLB (refer to Section 4.7.5, "Address Translation in Debug Mode"). In both cases, dseg addresses are unmapped and uncached.

Chapter 5

Exception Processing

This chapter describes CP0 exception processing and all exception conditions supported by the 5K microprocessor core. The CP0 registers used in exception processing are described in detail in Chapter 6, “Coprocessor 0 Registers.”

5.1 Overview

Exceptions are events which result in the suspension of the normal sequence of instruction execution and the transfer of control to another sequence of instructions that handles the exceptional condition. When an exception occurs, the relevant instruction and all those that follow it in the pipeline are cancelled, and a new instruction stream that handles the exception begins. Exceptions may be *precise* or *imprecise*. Precise exceptions, sometimes called *synchronous* exceptions, result from the execution of an instruction, for example, a TLB Invalid exception. Imprecise exceptions are caused by an earlier instruction in the instruction flow or by event unconnected to instruction execution, for example hardware interrupts.

When the CP0 detects an exceptional condition, the current sequence of instruction execution is suspended and the processor enters Kernel or Debug Mode, disables interrupts, and starts fetching instructions from one of the predefined exception-handler addresses (known as the exception *vector*). The exception handler saves sufficient processor state information in CP0 registers to resume program execution after the exception has been handled, including the current operating mode, the masking of interrupts, and the PC of the instruction where execution can be resumed.

There are two types of exceptions supported by the 5K architecture: *normal* exceptions and *debug* exceptions. Normal exceptions are part of the normal program flow, while debug exceptions are caused by the occurrence of a condition resulting from the debugging process, for example, a predefined break condition. In this chapter, normal exceptions will be described first, followed by a description of the debug exceptions.

Table 5-1 contains a list and a brief description of all exception conditions. The exceptions are listed in the order of their relative priority, from highest priority (Reset/EJTAG Boot) to lowest priority (TLB Modified - Data Access).

Table 5-1 Priority of Exceptions

Exception	Description
EJTAG Boot	Simultaneous assertion of EJ_DINT and one of the reset signals (SI_ColdReset or SI_Reset).
Reset	Assertion of SI_ColdReset.
Soft Reset	Assertion of SI_Reset.
Debug Single Step	Debug Single Step.
Debug Interrupt	Assertion of EJ_DINT.
Debug Data Break on Load Imprecise	Debug Data Break on Load Imprecise (Data Break on Load with address + value match).
Non Maskable Interrupt (NMI)	Asserting edge of SI_NMI detected.
Cache Error - Data access	Cache error on a load or store data reference (imprecise).
Machine Check	TLB write which would cause multiple matching entries in the TLB (imprecise).

Table 5-1 Priority of Exceptions (Continued)

Exception	Description
Data Bus Error	Bus error on a load or store data reference (imprecise).
Interrupt	Assertion of enabled hardware or software interrupt.
Deferred Watch	Deferred Watch (unmasked by EXL and ERL being cleared or exiting Debug Mode).
Debug Instruction Hardware Break	Debug Instruction Hardware Break.
Watch - instruction fetch	Watch address match detected on an instruction fetch.
Address Error - instruction fetch	Instruction fetch with address alignment error or instruction fetch from protected or invalid address area.
TLB/XTLB Refill - instruction fetch	TLB miss on instruction fetch.
TLB Invalid - instruction fetch	The Valid bit set to zero in the TLB entry matching the address referenced by an instruction fetch.
Cache Error - instruction fetch	Cache error on an instruction fetch.
Instruction Bus Error	Bus error on an instruction fetch.
Debug Breakpoint	Debug Breakpoint. Execution of SDBBP instruction.
Instruction Validity exceptions	CpU, MDMX, RI: An instruction could not be completed because it was not allowed access to the required resources, or it was illegal: Coprocessor Unusable (CpU or MDMX), Reserved Instruction (RI). If both exceptions occur on the same instruction, the Coprocessor Unusable exception has higher priority.
Execution exception	Sys: Execution of SYSCALL instruction.
	Bp: Execution of BREAK instruction.
	RI: Execution of a Reserved Instruction.
	Ov: Execution of an arithmetic instruction that overflows.
	Tp: Execution of a trap (when trap condition is true).
	FPE and C2E: Floating-point exception and COP2 exception from optional attached coprocessor(s).
Debug Data Break, load or store	Debug Data Break on Load (address only) or Debug Data Break on Store (address only or address + data value).
Watch - data access	Reference to an address matching one of the watch registers (data).
Address Error - load	Data load address alignment error. Data load reference from protected or invalid address area.
Address Error - store	Store address alignment error. Store to protected or invalid address area.
TLB/XTLB Refill - data access	TLB miss on a data access.
TLB Invalid - data access	The valid bit set to zero in the TLB entry matching the address referenced by a load or store instruction.
TLB Modified - data access	The Dirty bit set to zero in the TLB entry matching the address referenced by a store instruction.
Cache Error - instruction cache	Cache error detected in the instruction cache by the CACHE instruction.

5.1.1 Interrupt and NMI Latency

Some instructions cannot be cancelled by the occurrence of any exception other than a TLB exception after they have entered the Memory Fetch (M) stage of the pipeline. Thus, if there is a slip in the M stage of one of these instructions, the time required to respond to an interrupt will increase. (Refer to Section 2.9, "Slip Conditions and Interlock Handling".) In this case, interrupt-latency calculations must take into account any stall due to transactions on the processor bus. These instructions include all loads and stores, PREF(X), and CACHE.

Not all exceptions need be considered in latency calculations. For example, the Bus/Cache Errors on data access indicate catastrophic system conditions, and the exceptions used in debugging, such as Debug Interrupt and Debug Data Break on Load Imprecise, only occur in Debug Mode.

5.1.2 Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location 0xFFFF.FFFF.BFC0.0000. EJTAG Debug exceptions are vectored to location 0xFFFF.FFFF.BFC0.0480 if *Debug Control*$_{ProbTrap}$ = 0; otherwise, they are vectored to 0xFFFF.FFFF.FF20.0200. Note that while in Debug Mode, the block of addresses starting at the EJTAG vector (0xFFFF.FFFF.FF20.0200) are mapped to the TAP interface, and not to system memory (refer to Chapter 10).

Vector addresses for all other exceptions are a combination of a vector offset and a base address. Table 5-2 specifies the vector base address for each exception, taking into account the state of the *Status* register's *BEV* bit. Table 5-3 specifies the exception base-address offsets, and Table 5-4 provides the vector addresses.

Table 5-2 Exception Vector Base Addresses

Exception	**Status$_{BEV}$**	
	0	**1**
Reset, Soft Reset, NMI	0xFFFF.FFFF.BFC0.0000	
Debug (with *Debug Control*$_{ProbTrap}$ = 0)	0xFFFF.FFFF.BFC0.0480	
Debug (with *Debug Control*$_{ProbTrap}$ = 1)	0xFFFF.FFFF.FF20.0200	
Cache Error	0xFFFF.FFFF.A000.0000	0xFFFF.FFFF.BFC0.0200
Other	0xFFFF.FFFF.8000.0000	0xFFFF.FFFF.BFC0.0200

Table 5-3 Exception Vector Offsets

Exception	**Vector Offset**
TLB Refill, EXL = 0	0x000
64-bit XTLB Refill, EXL = 0	0x080
Cache error	0x100
Interrupt, Cause$_{IV}$ = 1	0x200
General Exception	0x180
Reset, Soft Reset, NMI	None (Uses Reset Base Address)

Table 5-4 Exception Vectors

Exception	BEV	EXL	IV	*Debug Control* [*ProbTrap*]	Vector Address
Reset, Soft Reset, NMI	x[a]	x	x	x	0xFFFF.FFFF.BFC0.0000
Debug	x	x	x	0	0xFFFF.FFFF.BFC0.0480
Debug	x	x	x	1	0xFFFF.FFFF.FF20.0200
TLB Refill	0	0	x	x	0xFFFF.FFFF.8000.0000
XTLB Refill	0	0	x	x	0xFFFF.FFFF.8000.0080
TLB Refill	0	1	x	x	0xFFFF.FFFF.8000.0180
XTLB Refill	0	1	x	x	0xFFFF.FFFF.8000.0180
TLB Refill	1	0	x	x	0xFFFF.FFFF.BFC0.0200
XTLB Refill	1	0	x	x	0xFFFF.FFFF.BFC0.0280
TLB Refill	1	1	x	x	0xFFFF.FFFF.BFC0.0380
XTLB Refill	1	1	x	x	0xFFFF.FFFF.BFC0.0380
Cache Error	0	x	x	x	0xFFFF.FFFF.A000.0100
Cache Error	1	x	x	x	0xFFFF.FFFF.BFC0.0300
Interrupt	0	0	0	x	0xFFFF.FFFF.8000.0180
Interrupt	0	0	1	x	0xFFFF.FFFF.8000.0200
Interrupt	1	0	0	x	0xFFFF.FFFF.BFC0.0380
Interrupt	1	0	1	x	0xFFFF.FFFF.BFC0.0400
All others	0	x	x	x	0xFFFF.FFFF.8000.0180
All others	1	x	x	x	0xFFFF.FFFF.BFC0.0380

a. An x denotes “don’t care”.

5.1.3 EPC, ErrorEPC, and DEPC Values

For those exceptions that cause the *EPC*, *ErrorEPC*, or *DEPC* register to be loaded with the address at which processing can be resumed after the exception has been handled, the address depends on whether or not the current instruction is in the delay slot of a branch or jump instruction: if the current instruction is in a delay slot, the new value in the *EPC*, *ErrorEPC*, or *DEPC* register is the PC of the preceding jump or branch instruction; if not, the new value is the PC of the current instruction. When the *EPC* or *DEPC* register is loaded for an instruction in a branch delay slot, $Cause_{BD}$ or $Debug_{DBD}$ respectively is set; otherwise, the bit is cleared.

5.1.4 General Exception Processing

With the exception of the Reset, Soft Reset, NMI, and Debug-Mode exceptions, each of which has its own special processing described later in this chapter, all exception conditions are processed as follows:

- If the *EXL* bit in the *Status* register is zero, the *EPC* register is loaded with the PC at which execution may be resumed, and the C*ause* register’s *BD* bit is set to the appropriate value. (The specific value for the *EPC* register is described in Section 5.1.3, "EPC, ErrorEPC, and DEPC Values") If the *EXL* bit in the *Status* register is set to one, the *EPC* register is not loaded, and the *BD* bit is not changed.

- The *Coprocessor Exception* (*CE)* bit and *ExcCode* fields of the *Cause* register are loaded with the values appropriate to the exception. The *CE* field is loaded for all non-debug exceptions that update the *Cause* register, but is only defined for the Coprocessor Unusable exception.
- The *EXL* bit is set in the *Status* register.
- Processing is started at the exception vector (refer to Table 5-1).

The value loaded into *EPC* is the return address for the exception, and in most cases does not need to be modified by exception-handler software. Nor, in most cases, does software have to read the value of the *Cause* register's *BD* bit. However, in the case of precise exceptions, software can combine the information provided in *EPC* and *BD* to identify the address of the instruction that caused the exception.

Individual exception types may load additional information into other CP0 registers, and this is noted in the description of each individual exception in later sections of this chapter.

The basic sequence of general exception-processing operations is summarized below.

```
                 if $Status_{EXL}$ = 0
          if InstructionInBranchDelaySlot then
             EPC <- PC of branch
             $Cause_{BD}$ <- 1
          else
             EPC <- PC of instruction
             $Cause_{BD}$ <- 0
          endif
          if ExceptionType = TLBRefill then
             vectorOffset <- 0x000
          elseif ExceptionType = XTLBRefill then
             vectorOffset <- 0x080
          elseif (ExceptionType = Interrupt) and
              ($Cause_{IV}$ = 1) then
             vectorOffset <- 0x200
          else
             vectorOffset <- 0x180
          endif
else
          vectorOffset <- 0x180
       endif
   $Cause_{CE}$ <- FaultingCoprocessorNumber
   $Cause_{ExcCode}$ <- ExceptionType
$Status_{EXL}$ <- 1
if $Status_{BEV}$ = 1 then
          PC <- 0xFFFF.FFFF.BFC0.0200 + vectorOffset
else
          PC <- 0xFFFF.FFFF.8000.0000 + vectorOffset
endif
```

5.2 Reset Exception

A Reset exception occurs when a hard reset is signaled to the processor by the assertion of *SI_ColdReset* while *EJ_DINT* is deasserted. 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 is 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* and *Config1* registers are initialized with their reset state.
- The *BEV, TS, SR, NMI,* and *ERL* fields of the *Status* register are initialized to a specified state.
- The *ErrorEPC* register is loaded as described in Section 5.1.3, "EPC, ErrorEPC, and DEPC Values" Note that this values may or may not be predictable.
- The *Debug* register is initialized as documented in Section 6.20, "Debug Register (CP0 Register 23, Select 0)"
- Processing is started at the Reset exception vector.

Cause Register ExcCode Value

None

Additional State Saved

None

Exception Vector Used

Reset

Operation

```
Random <- TLBEntries - 1
Wired <- 0
Config <- ConfigurationState
$Config_{K0}$ <- 2
Config1 <- ConfigurationState
    $Status_{BEV}$ <- 1
    $Status_{TS}$ <- 0
    $Status_{SR}$ <- 0
    $Status_{NMI}$ <- 0
    $Status_{ERL}$ <- 1
    $Status_{RP}$ <- 0
$WatchLo_{I}$ <- 0
    $WatchLo_{R}$ <- 0
    $WatchLo_{W}$ <- 0
    if InstructionInBranchDelaySlot then
            ErrorEPC <- PC of branch
    else
            ErrorEPC <- PC of current instruction
    endif
    PC <- 0xFFFF.FFFF.BFC0.0000
```

5.3 Soft Reset Exception

A Soft Reset exception occurs when the soft reset is signaled to the processor by the assertion of *SI_Reset*. This exception is not maskable. When a soft reset exception occurs, the processor performs a subset of the full reset initialization, as described below.

A Soft Reset exception does not unnecessarily change the state of the processor, but does 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 as described in Section 5.1.3, "EPC, ErrorEPC, and DEPC Values" Note that this value may or may not be predictable.
- The *Debug* register is initialized as documented in Section 6.20, "Debug Register (CP0 Register 23, Select 0)"
- Processing is started at the Reset exception vector.

Cause Register ExcCode Value

None

Additional State Saved

None

Exception Vector Used

Reset

Operation

```
$Status_{BEV}$ <- 1
$Status_{TS}$ <- 0
$Status_{SR}$ <- 1
$Status_{NMI}$ <- 0
$Status_{ERL}$ <- 1
if InstructionInBranchDelaySlot then
        ErrorEPC <- PC of branch
else
        ErrorEPC <- PC of current instruction
endif
PC <- 0xFFFF.FFFF.BFC0.0000
```

5.4 Non-maskable Interrupt (NMI) Exception

A Non-maskable interrupt exception occurs when the processor detects a rising edge of the NMI signal, *SI_NMI*. An NMI exception is masked when the processor is in Debug Mode, or when the NMI Enable (*NMIE*) bit in the *Debug Control* register is set to zero.

An NMI occurs only on instruction boundaries, so it does not perform any reset or other hardware initialization. The state of the cache, memory, and other processor state is consistent. The contents of 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 as described in Section 5.1.3, "EPC, ErrorEPC, and DEPC Values".
- Processing is started at the Reset exception vector.

Note that one of the features of EJTAG debug is to allow postponing of the NMI exception by clearing the *Debug Control NMIE* bit. This does not mask the edge detection on the *SI_NMI* signal, but it defers the exception until the *Debug* $Control_{NMIE}$ bit is set. For further information, refer to Section 10.2.6, "Interrupts and NMIs".

Cause Register ExcCode Value

None

Additional State Saved

None

Exception Vector Used

Reset

Operation

```
if DebugControl$_{NMIE}$ = 1 then
          Status$_{BEV}$ <- 1
          Status$_{TS}$ <- 0
          Status$_{SR}$ <- 0
          Status$_{NMI}$ <- 1
          Status$_{ERL}$ <- 1
          if InstructionInBranchDelaySlot then
             ErrorEPC <- PC of branch
          else
             ErrorEPC <- PC of current instruction
          endif
          PC <- 0xFFFF.FFFF.BFC0.0000
else
          DeferNMIException
endif
```

5.5 Machine Check Exception

A Machine Check exception occurs when there is an attempt to execute a TLB write instruction which would cause multiple matching entries in the TLB. The *TS* bit in the *Status* register is set to indicate this condition. It is the responsibility of software to handle this exception, perhaps by flushing the entire TLB. If the condition can be corrected, software must clear the TLB Shutdown (*TS)* bit in the *Status* register before resuming normal operation. In Debug Mode, this exception is deferred when the *Imprecise Error Exception Inhibit* bit in the *Debug* register is set; the Machine Check exception occurs when the *IEXI* bit is cleared or when software return from Debug Mode.

Since the multiple match condition is detected during a TLB write, the processor will preserve the entry already in the TLB.

Cause Register ExcCode Value

MCheck

Additional State Saved

None

Exception Vector Used

General exception vector, offset = 0x180

Operation

See Section 5.1.4, "General Exception Processing".

5.6 Address Error Exception

An address error exception occurs when there is an attempt to perform any of the following operations:

- Instruction fetch, load, or store of a word that is not aligned on a word boundary
- Load or store of a doubleword that is not aligned on a doubleword boundary
- Aligned load or store of a word that is not aligned on a word boundary

- Load or store of a half-word that is not aligned on a half-word boundary
- Reference to the Kernel address space from Supervisor or User Mode
- Reference to the Supervisor address space from User Mode
- Reference an undefined, unimplemented, or disabled memory segment from any Mode

Cause Register ExcCode Value

ADEL: Reference was a load or an instruction fetch

ADES: Reference was a store

Additional State Saved

Register	Value
BadVAddr	Failing address
$\text{Context}_{\text{VPN2}}$	Not defined[a]
$\text{XContext}_{\text{VPN2}}$ $\text{XContext}_{\text{R}}$	Not defined[a]
$\text{EntryHi}_{\text{VPN2}}$ $\text{EntryHi}_{\text{R}}$	Not defined
EntryLo0	Not defined
EntryLo1	Not defined

[a] Software should not in any way depend on the fact that some of the bits of BadVAddr are also visible in this register.

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.7 TLB and XTLB Refill Exceptions

A TLB or XTLB (extended TLB) Refill exception occurs in a TLB-based MMU when none of the TLB entries match a reference to a mapped address space and the *EXL* bit in the *Status* register is zero. Note that this is distinct from the TLB Invalid exception, which occurs when a TLB entry matches, but the entry's Valid bit is zero.

The XTLB refill handler is used whenever a reference is made to a 64-bit mode memory region. The addressing mode of a region is controlled by the $Status_{KX}$ bit if the region is in Kernel address space, the $Status_{SX}$ bit if the region is in Supervisor address space, and by the $Status_{UX}$ bit if the region is in User address space. Refer to Chapter 4, "Memory Management," for a description of Kernel and User address spaces.

The TLB and XTLB Refill exceptions have distinct exception vector offsets: 0x000 for a TLB Refill, and 0x080 for an XTLB Refill.

Cause Register ExcCode Value

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

Additional State Saved

Register	Value
BadVAddr	Failing address
Context$_{VPN2}$	*VPN2* of failing address
XContext$_{VPN2}$ XContext$_{R}$	*R*/*VPN2* of failing address
EntryHi$_{VPN2}$ EntryHi$_{R}$	*R*/*VPN2* of failing address
EntryLo0	Not defined
EntryLo1	Not defined

Exception Vector Used

TLB Refill vector (general exception vector, offset 0x000) if access was to 32-bit address space and *Status*$_{EXL}$ = 0 at the time of exception.

XTLB Refill vector (general exception vector, offset 0x080) if access was to 64-bit address space and *Status*$_{EXL}$ = 0 at the time of exception.

General exception vector, offset 0x180, if *Status*$_{EXL}$ = 1 at the time of exception, regardless of address space

Operation

See Section 5.1.4, "General Exception Processing"

5.8 TLB Invalid Exception

A TLB Invalid exception occurs when a TLB entry in a TLB-based MMU matches a reference to a mapped address space, but the matched entry has the Valid bit set to zero.

Note that a TLB Invalid exception is indistinguishable from the condition in which no TLB entry matches a reference to a mapped address space and *Status*$_{EXL}$ = 1. Both use the general exception vector and supply an ExcCode value of TLBL or TLBS. The only way to distinguish these two cases is to probe the TLB for a matching entry, using the TLB Probe (TLBP) instruction.

Cause Register ExcCode Value

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

Additional State Saved

Register	Value
BadVAddr	Failing address
Context$_{VPN2}$	*VPN2* of failing address
XContext$_{VPN2}$ XContext$_R$	*R*/*VPN2* of failing address
EntryHi$_{VPN2}$ EntryHi$_R$	*R*/*VPN2* of failing address
EntryLo0	Not defined
EntryLo1	Not defined

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.9 TLB Modified Exception

A TLB Modified exception occurs in a TLB-based MMU when a store instruction references a mapped address, and the matching TLB entry has the Dirty bit cleared (indicating that the entry cannot be modified).

Cause Register ExcCode Value

Mod

Additional State Saved

Register	Value
BadVAddr	Failing address
Context$_{VPN2}$	*VPN2* of failing address
XContext$_{VPN2}$ XContext$_R$	*R*/*VPN2* of failing address
EntryHi$_{VPN2}$ EntryHi$_R$	*R*/*VPN2*/*B* of failing address
EntryLo0	Unchanged
EntryLo1	Unchanged

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.10 Cache Error Exception

A Cache Error exception occurs when an instruction or data reference detects a cache error. Because the error is in a cache, the exception vector is an unmapped, uncached address. Note that the Cache Error exception is precise for instruction cache errors, and that the *ErrorEPC* register will contain either the PC of the instruction that actually caused the reference, or the immediately preceding jump or branch instruction, as described in Section 5.1.3, "EPC, ErrorEPC, and DEPC Values" However, cache errors in the data cache cause imprecise exceptions, and they are not guaranteed not to update memory or CP0 registers. While in Debug Mode, this exception is deferred when the *Imprecise Error Exception Inhibit* bit in the *Debug* register is set; the Cache Error exception occurs when the *IEXI* bit is cleared or when software returns from Debug Mode.

Cause Register ExcCode Value

N/A

Additional State Saved

Register	Value
CacheErr	Error state

Exception Vector Used

Cache error vector, offset 0x100.

Operation

```
    CacheErr <- ErrorState
    if InstructionInBranchDelaySlot then
          ErrorEPC <- PC of branch
    else
          ErrorEPC <- PC current instruction
    endif
if $Status_{BEV}$ = 1 then
          PC <- 0xFFFF.FFFF.BFC0.0200 + 0x100
else
          PC <- 0xFFFF.FFFF.A000.0000 + 0x100
endif
```

5.11 Bus Error Exception

A Bus Error exception occurs when an instruction or data access make a bus request (due to a cache miss or an uncacheable reference) and that request terminates with an error. Since the processor implements non-blocking (scheduled) loads, in most cases the *EPC* register will not contain the PC of the instruction which caused the bus transaction. While in Debug Mode, this exception is deferred when the *Imprecise Error Exception Inhibit* bit in the *Debug* register is set; the Bus Error exception occurs when the *IEXI* bit is cleared or when software returns from Debug Mode.

Cause Register ExcCode Value

IBE: Error on an instruction reference (precise)

DBE: Error on a data reference (imprecise)

Additional State Saved

None

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.12 Integer Overflow Exception

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

Exception Vector Used

General exception vector, offset 0x180.

Operation

See Section 5.1.4, "General Exception Processing"

5.13 Trap Exception

A Trap exception occurs when the condition tested by a trap instruction is TRUE.

Cause Register ExcCode Value

Tr

Additional State Saved

None

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.14 System Call Exception

A System Call exception occurs when a SYSCALL instruction is executed.

Cause Register ExcCode Value

Sys

Additional State Saved

None

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.15 Breakpoint Exception

A Breakpoint exception occurs when a BREAK instruction is executed.

Cause Register ExcCode Value

Bp

Additional State Saved

None

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.16 Reserved Instruction Exception

A Reserved Instruction exception occurs when a reserved or undefined major opcode or function field is executed.

For coprocessor instructions issued to the coprocessor interface, signaling of this exception is handled by the coprocessor.

This exception will also be signaled for instructions performing 64-bit operations when these are not enabled (by either Kernel or Debug Mode, or by $Status_{\mathbf{PX}}$ or $Status_{\mathbf{UX}}$ in User Mode)

Cause Register ExcCode Value

RI

Additional State Saved

None

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.17 Coprocessor Unusable Exception

A Coprocessor Unusable exception occurs when an attempt is made to execute a coprocessor instruction for either:

- A coprocessor that has not been marked usable by setting its *CU* bit in the *Status* register
- The CP0, when it has not been marked usable, and the processor is executing in User Mode

Cause Register ExcCode Value

CpU

Additional State Saved

Register	Value
Cause_{CE}	Unit number of the coprocessor being referenced

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.18 MDMX Coprocessor Unusable Exception

A MDMX Coprocessor Unusable exception occurs when an attempt is made to execute a MDMX instruction if either:

- No coprocessor implementing MDMX is attached to the processor
- The attached MDMX coprocessor has not been enabled by the MD bit in the Status register.

Cause Register ExcCode Value

MDMX

Additional State Saved

None.

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.19 Floating-Point Exception

The Floating-point exception is signaled by an attached FPU by signaling an FPE exception over the coprocessor interface. If no FPU is attached, the coprocessor exception signals must be tied to zero.

Cause Register ExcCode Value

FPE

Additional State Saved

If a FPU is attached, then refer to the documentation for that FPU.

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.20 Coprocessor 2 Exception

The Coprocessor 2 exception is signaled by an attached coprocessor by signaling an C2E exception over the coprocessor interface. If no coprocessor is attached the coprocessor exception signals must be tied to zero.

Cause Register ExcCode Value

C2E

Additional State Saved

If a coprocessor is attached, then refer to the documentation for that coprocessor.

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.21 Watch Exception

The watch exception occurs 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 in the *Status* register are both zero. If either bit is a one when the Watch exception would normally be taken, the Watch Pending bit (*WP)* in the *Cause* register is set, and the exception is deferred until both the *EXL* and *ERL* are zero. Software may use the *WP* bit in the *Cause* register to determine if the *EPC* register points to the instruction that caused the Watch exception, or if the exception occurred while in Kernel Mode. The *WP* bit directly causes a watch exception, so the exception handler must clear this bit in order to prevent a Watch exception loop when the handler completes.

Cause Register ExcCode Value

WATCH

Additional State Saved

Register	Value
Cause_{WP}	If set, this bit indicates that the Watch exception was deferred until both $Status_{EXL}$ and $Status_{ERL}$ were zero.

Exception Vector Used

General exception vector, offset 0x180

Operation

See Section 5.1.4, "General Exception Processing"

5.22 Interrupt Exception

The processor supports eight interrupt requests, which can be grouped into the following four categories:

- Software interrupts - Two software interrupt requests are made via software writes to the *IP0* and *IP1* bits in the *Cause* register.

- Hardware interrupts - Six hardware interrupt requests, numbered 0 through 5, are made via external requests to the processor. Hardware interrupts are level-sensitive, and an interrupt should hold its interrupt signal asserted until explicitly cleared by software.
- Timer interrupt - Hardware interrupt 5 can be used for the timer interrupt. Refer to the description of the *Count* and *Compare* registers in Chapter 6 for further details on how to use those registers to generate timer interrupts.
- Performance counter overflow - Hardware interrupt 5 is internally ORed with the overflow indication from the performance counters. For more details, refer to Section 6.22, "Performance Counter Register (CP0 Register 25, select 0-3)".

The current interrupt requests are visible at any time by reading the *IP* field in the *Cause* register (not just after an interrupt exception has occurred). The mapping of *Cause* register bits to the various interrupt requests is shown in Table 5-5.

Table 5-5 Mapping of Interrupts to the Cause and Status Registers

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

For hardware interrupts, *SI_Int*[0:5] are masked by the *IntE* bit of the *Debug Control* register before they are written to the *IP*2-*IP*7 bits of the *Cause* register.

For each bit in the *IP* field in the *Cause* register, there is a corresponding bit in the *IM* field of the *Status* register. An interrupt is taken only when all of the following are true:

- An interrupt request bit is a one in the *IP* field of the *Cause* register.
- The corresponding mask bit is a one in the *IM* field of the *Status* register. (The mapping of bits is shown in Table 5-5.)
- The *IE* bit in the *Status* register is a one.
- The *EXL* and *ERL* bits in the *Status* register are both zero.
- The *IntE* of the EJTAG *Debug Control Register* is set.

Logically, $Cause_{IP}$ is bitwise ANDed with $Status_{IM}$, the eight resulting bits are ORed together, and that value is ANDed with $Status_{IE}$. The final interrupt request is then signaled only if both $Status_{EXL}$ and $Status_{ERL}$are zero, corresponding to a non-exception, non-error processing mode.

Note that EJTAG is capable of masking interrupts in Non-Debug Mode.

Register ExcCode Value

Int

Additional Information

Register	Value
Cause_{IP}	Indicates the interrupts that are pending.

Exception Vector Used

General exception vector, offset 0x180, if the *IV* bit in the *Cause* register is zero.

General exception vector, offset 0x200, if the *IV* bit in the *Cause* register is one.

Operation

See Section 5.1.4, "General Exception Processing"

5.23 Debug Exceptions

The 5K processor supports the following six debug exceptions:

- Debug Single-Step (DSS)
- Debug Interrupt (DINT) with EJTAG Boot as a special case
- Debug Instruction Break (DIB)
- Debug Data Break on Load with address match only (DDBL)
- Debug Data Break on Store (DDBS)
- Debug Data Break on Load Imprecise with address and data match (DDBLImpr)
- Debug software Breakpoint (DBp)

5.23.1 Exception Handling of Debug Exceptions

All debug exceptions are processed as follows:

- The *DEPC* register and the *DBD* bit of the *Debug* register are updated as specified in Section 5.1.3, "EPC, ErrorEPC, and DEPC Values".
- The *Debug* register bits *DSS*, *DINT*, *DIB*, *DDBL*, *DDBS*, *DDBLImpr*, and *DBp* are updated appropriately, depending on the debug exception.
- *DExcCode* field in the *Debug* register is undefined.
- The *Halt* and *Doze* bits in the *Debug* register are updated to reflect the state of the hardware at the time of the exception.
- *IEXI* bit in *Debug* register is set to inhibit imprecise exceptions in the start of the debug handler.
- The *DM* bit in the *Debug* register is set to enter Debug Mode.
- Processing is started at the debug exception vector.

Cause Register ExcCode Value

N/A

Additional State Saved

None

Note that *DDBLImpr* may be set when other debug exception bits are also set, since it indicates a one-time event, possibly triggered by an instruction which has completed prior to the occurrence of the other debug exception. For the same reason, *DDBLImpr* may also be set by the processor after Debug Mode has been entered. The debug exception handler should execute a SYNC instruction as part of its entry code to ensure that *DDBLImpr* has been updated with any outstanding imprecise debug data breakpoints.

Exception Vector Used

Debug exception vector (0xFFFF.FFFF.BFC0.0480 or 0xFFFF.FFFF.FF20.0200).

Operation

```
if InstructionInBranchDelaySlot then
          $DEPC$ <- PC of branch
          $Debug_{DBD}$ <- 1
   else
          $DEPC$ <- PC current instruction
          $Debug_{DBD}$ <- 0
      endif
   $Debug$[DDBLImpr, DINT - DSS] <- DebugExceptionType
   $Debug_{Halt}$ <- HaltStatusAtDebugExceptionTime
      $Debug_{Doze}$ <- DozeStatusAtDebugExceptionTime
      $Debug_{DExcCode}$ <- UNPREDICTABLE
      $Debug_{IEXI}$ <- 1
          $Debug_{DM}$ <- 1
if $DebugControl_{ProbTrap}$ = 1 then
          PC <- 0xFFFF.FFFF.FF20.0200
eelse
          PC <- 0xFFFF.FFFF.BFC0.0480
ndif
```

5.23.2 Debug Breakpoint Exception

A Debug Breakpoint exception occurs when an SDBBP instruction is executed.

Debug Register Debug Status Bit Set

DBp

5.23.3 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 indicate the instruction that caused the instruction hardware breakpoint match. This exception can only occur if instruction hardware breakpoints are implemented.

Debug Register Debug Status Bit Set

DIB

5.23.4 Debug Data Break Load/Store Exception

A Debug Data Break Load exception occurs when a data hardware breakpoint with address match only matches the address of a load instruction. A Debug Data Break Store exception occurs when a data hardware breakpoint matches the address and optionally the store value of a store instruction. The *DEPC* register and *DBD* bit in the *Debug* register indicate the load/store instruction that caused the data hardware breakpoint match, as this is a precise debug exception. The load/store instruction that caused the debug exception has not completed (it has not updated the destination register or memory location), and the instruction is therefore executed on return from the debug handler. This exception can only occur if data hardware breakpoints with precise data breaks are implemented and will not occur for load breakpoints with value compare, since that exception is reported as an imprecise Debug Data Break Exception, as described in Section 5.23.5, "Debug Data Break Load Imprecise Exception".

Debug Register Debug Status Bit Set

DDBL (load) or DDBS (store)

5.23.5 Debug Data Break Load Imprecise Exception

A Debug Data Break Load Imprecise exception occurs when a data hardware breakpoint matches a load access of an executed load instruction, and it is not possible to take a precise debug exception on the instruction. This case occurs when a data hardware breakpoint was set up with a value compare. The *DEPC* register and the *DBD* bit in the *Debug* register indicate an instruction later in the execution flow instead of the load instruction that caused the data hardware breakpoint match. The *DDBLImpr* bit in the *Debug* register indicates that a Debug Data Break Load Imprecise exception occurred. The instruction that caused the Debug Data Break Load Imprecise exception will have completed. It updates its destination register and is not executed on return from the debug handler. This exception can only occur if data hardware breakpoints with imprecise data breakpoints are implemented.

Imprecise debug exceptions from data hardware breakpoints are indicated together with another debug exception if the load transaction that caused the data hardware breakpoint match did not complete until after another debug exception occurred. In this case, the other debug exception was the cause of entering Debug Mode, so the *DEPC* register and the *DBD* bit in the *Debug* register point to the instruction causing that exception.

A SYNC instruction must be executed by the debug exception handler before the *DDBLImpr* bit in the *Debug* register and the *BS*[*n*] bits for the data hardware breakpoint are read, in order to ensure that all imprecise breaks are resolved and the bits are fully updated. This scheme ensures that all breakpoints matching due to code executed before the debug exception are indicated by the *DDBLImpr* and *BS*[*n*] bits.

Debug Register Debug Status Bit Set

DDBLImpr

5.23.6 Debug Single Step Exception

When single-step mode is enabled, a Debug Single Step exception is taken for the second execution step in Non-Debug Mode. An execution step is a single instruction, or an instruction pair consisting of a jump/branch instruction and the instruction in the associated delay slot. The *SSt* bit in the *Debug* register enables Debug Single Step exceptions.

The *DEPC* register points to the instruction on which the Debug Single Step exception occurred, which will also be the next instruction to execute when returning from Debug Mode. The debug software can examine the system state before this instruction is executed. Thus the *DEPC* will not point to the instruction(s) that have just executed in the execution step, but rather to the instruction following the execution step. The Debug Single Step exception never occurs on an instruction in a jump/branch delay slot. A jump/branch and the instruction in the delay slot are always executed in one execution step. The *DBD* bit in the *Debug* register can therefore never be set for a Debug Single Step exception.

The Debug Single Step exceptions will be taken on the second execution step following the return to Non-Debug Mode, regardless of whether the first execution step caused a precise non-debug exception. If this was the case the Debug Single Step exception will be taken on the first instruction in the exception handler for the non-debug exception.

Debug exceptions are unaffected by single-step mode; returning to an SDBBP instruction with single step enabled causes a Debug Breakpoint exception with the *DEPC* register pointing 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* register pointing to the SDBBP instruction.

To ensure proper functionality of single-step execution, the Debug Single Step exception has priority just below Soft Reset.

Debug Register Debug Status Bit Set

DSS

5.23.7 Debug Interrupt Exception

The Debug Interrupt exception is an imprecise debug exception that is taken as soon as possible, but has no specific relation to the executed instructions. The *DEPC* register and the *DBD* bit in the Debug register reference the instruction at which execution can be resumed after Debug Interrupt exception service.

Debug interrupt requests are ignored when the processor is in Debug Mode, and pending requests are cleared when the processor takes any debug exception, including debug exceptions other than Debug Interrupt exceptions.

If the processor is in low-power mode with the internal clock stopped due to the execution of a WAIT instruction, a Debug Interrupt will restart the clock and the processor pipeline.

Debug Register Debug Status Bit Set

DINT

The following sources can cause Debug Interrupt exceptions:

- The EJ_DINT signal from the probe
  The EJ_DINT signal can request a debug interrupt on a low (0) to high (1) transition.
  The *EjtagBrk* bit in the *Debug Control* register provides similar DINT functionality from the probe, but with higher latency.
- The *EjtagBrk* Bit in the *Debug Control* Register
  The *EjtagBrk* bit in the *Debug Control* register requests a Debug Interrupt exception when set. Refer to Section 10.3, "Debug Control Register"
- A debug boot by EJTAGBOOT
  The EJTAGBOOT feature allows the reset initialization of a Reset or Soft Reset exception to be combined with a Debug Interrupt exception, so that the processor will fetch the first instruction from the Debug Exception Vector and not from the Reset Vector.

5.23.8 Handling of Exceptions in Debug Mode

Some exceptions are handled differently in Debug Mode. Table 5-6 specifies these differences.

Table 5-6 Exceptions In Debug Mode

Exception	Functionality In Debug Mode
Reset, Soft Reset	Identical to Non-Debug Mode in all respects.
Non-maskable Interrupt (NMI), Interrupt, Deferred Watch, Watch	Ignored. However, the edge detection for NMI is still working, but the exception is deferred until the processor has left Debug Mode.

Table 5-6 Exceptions In Debug Mode (Continued)

Exception	Functionality In Debug Mode
Machine Check, Address Error, TLB exceptions, Cache Error, Bus Error, execution-based exceptions	Exception taken, but handling differs from Non-Debug Mode: Only *Debug* and *DEPC* registers are updated by the exception; all other CP0 registers are unaffected. Debug exception vector is used for servicing.
SDBBP	Identical to the BREAK instruction (sets *DExcCode* to *Bp*).
All other exceptions	Ignored.

On TLB exceptions in Debug Mode, only the *Debug* and *DEPC* registers are updated, which makes TLB exceptions in Debug Mode more difficult to handle. However, their intended use in Debug Mode is to catch errors in debug software, and not to be *handled*.

For the handling of the Reset and Soft Reset exceptions in Debug Mode, refer to Section 5.2, "Reset Exception" and Section 5.3, "Soft Reset Exception".

All other handled exceptions in Debug Mode are processed as follows:

- The *DEPC* register and the *DBD* bit of the *Debug* register are updated following the rules in Section 5.1.3, "EPC, ErrorEPC, and DEPC Values".
- *DSS*, *DINT*, *DIB*, *DDBL*, *DDBS*, *DDBLImpr*, and *DBp* are cleared.
- The *DExcCode* in the *Debug* register. Note that the SDBBP instruction is treated exactly like the BREAK instruction in Debug Mode.
- The *Halt* and *Doze* bits in the *Debug* register are not defined following the exception.
- Processing is started at the debug exception vector.
- *IEXI* is set.

Cause Register ExcCode Value

N/A

Additional State Saved

None.

Exception Vector Used

Debug exception vector (0xFFFF.FFFF.BFC0.0480 or 0xFFFF.FFFF.FF20.0200).

Operation

```
if InstructionInBranchDelaySlot then
          DEPC <- PC of branch
          $Debug_{DBD}$ <- 1
   else
          DEPC <- PC current instruction
          $Debug_{DBD}$ <- 0
      endif
   Debug[DINT] <- 1
   Debug[DDBLImpr, DIB - DSS] <- 0
   $Debug_{DExcCode}$ <- ExcCode.
   $Debug_{Halt}$ <- NotDefined
      $Debug_{Doze}$ <- NotDefined
      if $DebugControl_{ProbTrap}$ = 1 then
```

```
          PC <- 0xFFFF.FFFF.FF20.0200
else
          PC <- 0xFFFF.FFFF.BFC0.0480
endif
```

Note that in systems which do not use the EJTAG debug solution, execution of an SDBBP instruction will nevertheless cause a debug exception. In this case, the debug exception should be viewed as a Reserved Instruction exception. Software can return from Debug Mode by executing a DERET instruction after updating the *DEPC* register with the address where normal execution is to be resumed.

5.23.9 EJTAG Boot

The EJTAG Boot exception is a combination of two exceptions: a reset exception (Hard or Soft Reset) and the Debug Interrupt exception. This exception is not maskable, but can only occur when enabled through the Test Access Port (TAP). All reset initialization is performed as specified in Section 5.2, "Reset Exception" In addition, *DEPC* is initialized to point to the reset vector (0xFFFF.FFFF.BFC0.0000) and the *Debug* register is set as if for a Debug Interrupt exception.

Cause Register ExcCode Value

None

Additional State Saved

None

Exception Vector Used

Debug (0xFFFF.FFFF.FF20.0200)

Operation

```
Perform CP0 register initializtion according to the reset (Reset or Soft Reset);
do not fetch from Reset vector;
   DEPC <- 0xFFFF.FFFF.BFC0.0000;
Take DINT exception.
```

Coprocessor 0 Registers

This chapter describes the Coprocessor 0 (CP0) registers. CP0 registers are summarized in Table 6-1 and described individually in the remaining sections of this chapter. The read/write properties of register bit fields are specified in Table 6-2.

Table 6-1 Coprocessor 0 Register Summary

Register Number	Sel	Register Name	Function
0	0	Index[a]	Index into the TLB array
1	0	Random[a]	Randomly generated index into the TLB array
2	0	EntryLo0[a]	Low-order portion of the TLB entry for even-numbered virtual pages
3	0	EntryLo1[a]	Low-order portion of the TLB entry for odd-numbered virtual pages
4	0	Context[a]	Pointer to page table entry in memory
5	0	PageMask[a]	Control for variable page size in TLB entries
6	0	Wired[a]	Controls the number of fixed (“wired”) TLB entries
8	0	BadVAddr	Reports the address for the most recent address-related exception
9	0	Count	Processor cycle count
10	0	EntryHi[a]	High-order portion of the TLB entry
11	0	Compare	Timer interrupt control
12	0	Status	Processor status and control
13	0	Cause	Cause of last general exception
14	0	EPC	Program counter at last exception
15	0	PRId	Processor identification and revision
16	0	Config	Configuration register
16	1	Config1	Configuration register 1
18	0	WatchLo	Low-order watchpoint address
19	0	WatchHi	High-order watchpoint address
20	0	XContext[a]	Extended-addressing page table context
23	0	Debug	Debug register
24	0	DEPC	Program counter at exception entering Debug Mode
25	0-3	PerfCnt	Performance counter interface
26	0	ErrCtl	Parity/ECC error control and status

Table 6-1 Coprocessor 0 Register Summary (Continued)

Register Number	Sel	Register Name	Function
27	0	CacheErr	Cache parity error control and status
28	0	TagLo	Low-order portion of cache tag interface
28	1	DataLo	Low-order portion of cache data interface
29	0	TagHi	High-order portion of cache tag interface
29	1	DataHi	High-order portion of cache data interface
30	0	ErrorEPC	Program counter at last error
31	0	DESAVE	Debug Exception Save Register

[a] This register is only used with the TLB-based MMU.

Table 6-2 Read/Write Properties

Read/Write Notation	Hardware Interpretation	Software Interpretation
R/W	A field in which all bits are readable and writable by hardware and by software. Hardware updates of this field are visible by a software read and software updates of this field are visible by a hardware read. If the Reset state of this field is specified as *Undefined*, either software or hardware must initialize the value before a read will return a predictable value. (This should not be confused with **UNDEFINED** behavior of the processor.)	
R/W1	A field in which all bits are readable and writable by hardware and readable and writable to 1 by software. Hardware updates of this field are visible by a software read and software updates of this field are visible by a hardware read. If the Reset state of this field is specified as *Undefined*, either software or hardware must initialize the value before a read will return a predictable value. (This should not be confused with **UNDEFINED** behavior of the processor.)	
R	A field which is either static or is updated only by hardware. If the Reset state of this field is either zero or *Preset*, hardware initializes this field to zero or to the appropriate state, respectively, on power-up. 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 in which the value written by software is ignored by hardware; that is, 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 that is done under the conditions specified in the description of the field.
0	A field which hardware does not update, and for which hardware can assume a zero value.	A field in 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 processor. Software reads of this field return zero if all previous software writes are zero. If the Reset state of this field is *Undefined*, software must write a zero to this field before it is guaranteed to read as zero.

6.1 Index Register (CP0 Register 0, Select 0)

Note: This register is only used with a TLB-based MMU.

The *Index* register is a 32-bit read/write register which contains the index used to access the TLB by the TLBP, TLBR, and TLBWI instructions.

Note that 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.

Figure 6-1 shows the format of the *Index* register; Table 6-3 describes the *Index* register fields.

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

Figure 6-1 Index Register

Table 6-3 Index Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	**Bits**			
P	31	Probe Failure. Set to 1 when the previous TLB Probe instruction (TLBP) failed to find a match in the TLB.	R	Undefined
Index	5:0	Index to the TLB entry used by the TLB Read (TLBR) and TLB Write Index (TLBWI) instructions.	R/W	Undefined
0	30:6	Must be written as zero; returns zero on read.	0	0

6.2 Random Register (CP0 Register 1, Select 0)

Note: This register is only used with a TLB-based MMU.

The *Random* register is a read-only register whose value is used by the TLBWR instruction to index an entry in the TLB.

The value of this 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 (as specified by 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 Indexed operation.
- An upper bound is set by the total number of TLB entries minus 1.

This register is implemented as a down-counter which is updated following the execution of a TLBWR instruction. The decrement is randomized by a Linear Feedback Shift Register (LFSR).

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

Figure 6-2 shows the format of the *Random* register; Table 6-4 describes the *Random* register fields.

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

Figure 6-2 Random Register

Table 6-4 Random Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
Random	5:0	TLB Random Index	R	TLB Entries - 1
0	31:6	Must be written as zero; returns zero on read.	0	0

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

Note: *EntryLo0* and *EntryLo1* are only used with a TLB-based MMU.

The pair of *EntryLo* registers act as the interface between the TLB and the TLBR, TLBWI, and TLBWR instructions. *EntryLo0* contains the entries for even pages, and *EntryLo1* contains the entries for odd pages.

The contents of these registers are only updated by hardware for a TLBR instruction and are undefined following TLB and Address Error exceptions.

Figure 6-3 shows the format of the *EntryLo* registers; Table 6-5 describes the *EntryLo* register fields.

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

Figure 6-3 EntryLo0, EntryLo1 Register

Table 6-5 32-bit EntryLo0, EntryLo1 Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
PFN	29:6	Page Frame Number. Corresponds to bits [35:12] of the physical address.	R/W	Undefined
C	5:3	Coherency attribute of the page. See Table 6-6.	R/W	Undefined
D	2	Dirty bit. Indicates 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
V	1	Valid bit. Indicates that the TLB entry, and thus the virtual page mapping, is 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

Table 6-5 32-bit EntryLo0, EntryLo1 Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bits			
G	0	Global bit. On a TLB write, the logical AND of the *G* bits from both *EntryLo0* and *EntryLo1* 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
0	31:30	Ignored on write; returns zero on read.	R	0

Table 6-6 specifies the encoding of the *C* field of the *EntryLo0* and *EntryLo1* registers and the *K0* field of the *Config* register. Note that the operation of the processor is **UNDEFINED** if software specifies an unimplemented encoding.

Table 6-6 Cache Coherency Attributes

C Value	Cache Coherency Attribute
0	Cacheable, noncoherent, write through, no write-allocate
1	Cacheable, noncoherent, write through, write-allocate
2	Uncached (write-around)
3-6	Cacheable, noncoherent, write-back (write-allocate)
7	Uncached accelerated

6.4 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. When there is 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 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 read only by the operating system and is not written by hardware.

The *BadVPN2* field is updated on TLB exceptions and undefined following Address Error exceptions.

Figure 6-4 shows the format of the *Context* Register; Table 6-7 describes the *Context* register fields.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
PTEBase																															

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PTEBase									BadVPN2																			0			

Figure 6-4 Context Register

Table 6-7 Context Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
PTEBase	63:23	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 contains bits 31:13 of the virtual address in the *BadVAddr* register. It is updated on TLB exceptions and undefined following Address Error exceptions.	R	Undefined
0	3:0	Must be written as zero; returns zero on read.	0	0

6.5 PageMask Register (CP0 Register 5, Select 0)

Note: Only used with a TLB-based MMU.

The *PageMask* register is a read/write register used for reading and writing the PageMask field of a TLB entry. It contains a comparison mask that sets the variable page size for each TLB entry.

Figure 6-5 shows the format of the *PageMask* register; Table 6-8 describes the *PageMask* register fields. The operation of the processor for other values of the *PageMask* than those listed in this table is UNDEFINED.

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

Figure 6-5 PageMask Register

Table 6-8 PageMask Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
Mask	24:13	The Mask field is a bit mask in which a "1" bit 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 6-9 Values for the Mask Field of the PageMask Register

Page Size	Bit											
	24											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

Table 6-9 Values for the Mask Field of the PageMask Register (Continued)

Page Size	Bit											
	24											**13**
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**

6.6 Wired Register (CP0 Register 6, Select 0)

Note: Only used with a TLB-based MMU.

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 6-6. Wired entries are fixed, non-replaceable entries which are not overwritten by a TLBWR instruction. Wired entries can be overwritten by a TLBWI instruction.

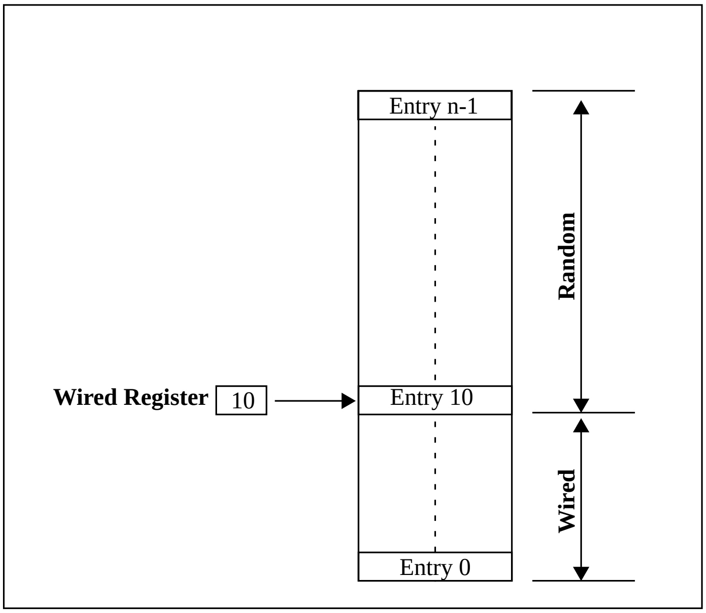


Figure 6-6 Wired and Random TLB Entries

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.

Figure 6-7 shows the format of the *Wired* register; Table 6-10 describes the *Wired* register fields.

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

Figure 6-7 Wired Register

Table 6-10 Wired Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
Wired	5:0	TLB wired boundary.	R/W	0
0	31:6	Must be written as zero; returns zero on read.	0	0

6.7 BadVAddr Register (CP0 Register 8, Select 0)

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

- Address error (AdEL or AdES)
- TLB Refill
- TLB Invalid (TLBL, TLBS)
- TLB Modified

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

Figure 6-8 shows the format of the *BadVAddr* register; Table 6-11 describes the *BadVAddr* register fields.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
BadVAddr																															

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

Figure 6-8 BadVAddr Register

Table 6-11 BadVAddr Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
BadVAddr	63:0	Virtual address that caused an exception.	R	Undefined

6.8 Count Register (CP0 Register 9, Select 0)

The *Count* register acts as a timer, incrementing by 1 on every other clock cycle, whether or not an instruction is executed, retired, or any forward progress is made through the pipeline.

The *Count* register can be written for functional or diagnostic purposes, including synchronization of processors and on Reset.

Figure 6-9 shows the format of the *Count* register; Table 6-12 describes the *Count* register fields.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">Count</td></tr>
</table>

Figure 6-9 Count Register

Table 6-12 Count Register Field Descriptions

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

6.9 EntryHi Register (CP0 Register 10, Select 0)

The *EntryHi* register contains the virtual address match information used by instructions that access the TLB (refer to Section 4.5, "TLB Management Instructions"

A TLB exception (TLB Refill, TLB Invalid, or TLB Modified) causes the mapped bits 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 a TLB match. The *ASID* field is not used when an FMT is connected to the MMU.

The *VPN2* and *R* fields of the *EntryHi* register are not defined following an Address Error exception.

Figure 6-10 shows the format of the *EntryHi* register; Table 6-13 describes the *EntryHi* register fields.

<table>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="2">R</td><td colspan="22">Fill</td><td colspan="8">VPN2</td></tr>
</table>

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="19">VPN2</td><td colspan="5">0</td><td colspan="8">ASID</td></tr>
</table>

Figure 6-10 EntryHi Register

Table 6-13 64-bit EntryHi Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
R	63:62	Virtual memory region, corresponding to $VA_{63:62}$. 00: xuseg (user address region) 01: xsseg (supervisor address region) 10: Reserved 11: xkseg (kernel address region)	R/W	Undefined
Fill	61:40	Fill bits. This field is reserved for expansion of the virtual address space. Returns zeros on read; ignored on write.	R	0
VPN2	39:13	$VA_{39:13}$ 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
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 writes and TLB comparison during address translation. If an FMT is connected to the MMU, this field must be written as zero; it returns zero on read.	R/W	Undefined
0	12:8	Must be written as zero; returns zero on read.	0	0

6.10 Compare Register (CP0 Register 11, Select 0)

The *Compare* register is used in conjunction with the *Count* register to implement a timer and a timer interrupt function. The *Compare* register maintains a constant value which does not change unless explicitly updated by software.

When the value of the *Count* register equals the value of the *Compare* register, the 5K processor core's *SI_TimerInt* output is asserted and continues to be asserted until the *Compare* register is written by software. This output can be routed back to the CPU through hardware interrupt 5 to set the Interrupt Pending bit (*IP*7) in the *Cause* register; when this bit is set, an interrupt will occur when the interrupt is enabled.

In normal use, the *Compare* register is write-only. However, for diagnostic purposes, it may be read and written.

Figure 6-11 shows the format of the *Compare* register; Table 6-14 describes the *Compare* register fields.

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="32">Compare</td></tr>
</table>

Figure 6-11 Compare Register

Table 6-14 Compare Register Field Descriptions

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

6.11 Status Register (CP 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 described below.

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

- *IE* = 1
- *EXL* = 0
- *ERL* = 0
- Debug_{DM} = 0

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

Operating Modes: The 5K supports four operating modes: Kernel, Supervisor, User, and a special Debug Mode used for EJTAG debugging. The encoding of the processor's operating mode is shown in Table 6-15. In the table, 'x' is used to denote a 'don't care' value. Do not use Status[KSU] = 11_2 under any circumstances.

Table 6-15 Processor Modes

Debug[DM]	Status[EXL]	Status[ERL]	Status[KSU]	Processor Mode
1	x	x	x	Debug
0	1	x	x	Kernel
0	x	1	x	Kernel
0	x	x	00_2	Kernel
0	0	0	01_2	Supervisor
0	0	0	10_2	User
x	x	x	11_2	UNDEFINED

In addition to the four modes of operation, bits in the Status register can selectively enable or disable the following operations:

- 64-bit addressing in user address space - controlled by the *UX* bit
- 64-bit addressing in supervisor address space - controlled by the *SX* bit
- 64-bit addressing in kernel address space - controlled by the *KX* bit
- 64-bit operations - controlled by the *PX* and *UX* bits in User Mode. Always enabled in all other modes.

When the *PX*, *UX*, *SX*, and *KX* bits are cleared, the processor is operating in MIPS32- compatibility mode. This mode guarantees that both user and operating-system code written for MIPS32 processors will execute correctly on the 5K microprocessor core.

Coprocessor Accessibility: The *Status* register's *CU* bits control coprocessor accessibility. If any coprocessor is unusable (its *CU* bit is set to zero), an instruction that accesses it generates an exception. Note that Coprocessor 0 is always enabled in Kernel and Debug Modes, regardless of the setting of the *CU0* bit.

Figure 6-12 shows the format of the *Status* register; Table 6-16 describes the *Status* register fields.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CU3-CU0				RP	FR	RE	MX	PX	BEV	TS	SR	NMI	0			IM7-IM0								KX	SX	UX	KSU		ERL	EXL	IE

Figure 6-12 Status Register

Table 6-16 Status Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
CU (CU3..CU0)	31:28	Coprocessor Usable. CU2..CU0 control access to coprocessor 2, 1, and 0 respectively. 0: Access to coprocessor not allowed 1: Access to coprocessor allowed Coprocessor 0 is always usable when the processor is in Kernel or Debug Mode, regardless of the state of the *CU0* bit. Coprocessor 1 and 2 can only be marked as usable if a coprocessor is actually attached to the CPU. For example, if no coprocessor 2 is attached, software cannot set *CU2*. Note that COP1X instructions are enabled by *CU1*. *CU3* is unused by the processor but is implemented as a read/write bit for backwards-compatibility. This bit can only be set to 1 if coprocessor 1 is attached to the CPU.	R/W	Undefined
RP	27	Reduced Power. Enables reduced-power mode. The state of the RP bit is available externally, via the *SI_RP* signal, for use by external logic to enable low-power mode. When a debug exception is taken, the *Debug* register's *Doze* bit is set by hardware to the current value of *RP*. 0: Low-power mode not enabled 1: Low-powered mode enabled	R/W	0
FR	26	Controls the floating-point register mode: 0: Floating-point registers can contain any 32-bit data type. 64-bit data types are stored in even-odd pairs of registers. 1: Floating-point registers can contain any data type Certain combinations of the FR bit and other state or operations can cause **UNPREDICTABLE** behavior. See the documentation for the floating-point processor for a discussion of these combinations.	R/W	Undefined
RE	25	Reverse Endian. Enables reverse-endian memory references in User Mode. 0: User Mode uses configured endianness 1: User Mode uses reversed endianness Neither Kernel Mode nor Supervisor Mode references are not affected by the state of this bit.	R/W	Undefined
MX	24	Enable access to MDMX™ resources on processors implementing MDMX. If a coprocessor which implements MDMX is not attached, the bit must be written as 0 and ignored on read.	R/W or R0	Undefined or 0

Table 6-16 Status Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bits			
PX	23	Processor Extension. Enables access to 64-bit registers and operations in User Mode, without enabling 64-bit addressing. 0: User Mode 64-bit registers and operations disabled 1: User Mode 64-bit registers and operations enabled	R/W	Undefined
BEV	22	Bootstrap Exception Vector. Controls the location of exception vectors. Refer to Table 5-4 in Chapter 5, “Exception Processing.” 0: Normal 1: Bootstrap	R/W	1
TS	21	TLB Shutdown. Set on Machine Check exceptions caused by an attempt to execute a TLB write instruction which would have caused a multiple match in the TLB. 0: No Machine Check exception 1: Machine Check exception NOTE: Software must clear this bit as part of the handling of a Machine Check exception; otherwise, a new Machine Check exception may be take by the processor when *ERL* and *ERL* are both 0.	R/W	0
SR	20	Soft Reset. 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	R/W	1 for Soft Reset; 0 otherwise
NMI	19	Non-maskable Interrupt. Indicates that the entry through the reset exception vector was due to an NMI. 0: Not NMI (Soft or Hard reset) 1: NMI	R/W	1 for NMI; 0 otherwise
IM (IM7..IM0)	15:8	Interrupt Mask. Used to individually mask the *Interrupt Pending* (*IP*) bits in the *Cause* register, in order to disable the corresponding interrupt exception. 0: Interrupt request disabled 1: Interrupt request enabled	R/W	Undefined
KX	7	Kernel Extension. If set, enables 64-bit addressing for memory references to Kernel address space. (64-bit registers and operations are always enabled in Kernel Mode.) 0: 64-bit addressing in Kernel address space disabled 1: 64-bit addressing in Kernel address space enabled	R/W[a]	Undefined
SX	6	Supervisor Extension. If set, enables 64-bit addressing for memory references to Supervisor address space.(64-bit registers and operations are always enabled in Supervisor Mode.) 0: 64-bit addressing in Supervisor address space disabled 1: 64-bit addressing in Supervisor address space enabled	R/W[a]	Undefined

Table 6-16 Status Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bits			
UX	5	User Extension. If set, enables 64-bit addressing for memory references to user address space and enables 64-bit registers and operations in User Mode. 0: 64-bit addressing in User address space disabled 1: 64-bit addressing in User address space enabled	R/W[a]	Undefined
KSU	4:3	The encoding of this field denotes the base operating mode of the processor as defined above. The encoding of this field is: 00_2: Base mode is Kernel Mode 01_2: Base mode is Supervisor Mode 10_2: Base mode is User Mode 11_2: Reserved. The operation of the processor is **UNDEFINED** if this value is written to the KSU field	R/W	Undefined
ERL	2	Error Level. Set by the processor when a Reset, Soft Reset, NMI, or Cache Error exception is taken. 0: Non-error level 1: Error level When *ERL* is set: • The processor is running in Kernel Mode (provided that *Debug*[*DM*] = 0). • Interrupts are disabled. • The ERET instruction will use the return address in *ErrorEPC* instead of *EPC*. • kuseg is treated as an unmapped and uncached region, which allows main memory to be accessed in the presence of cache errors.	R/W	1
EXL	1	Exception Level. Set by the processor when any exception other than Reset, Soft Reset, NMI or Cache Error exception is taken. 0: Non-exception level 1: Exception level When *EXL* is set: • The processor is running in Kernel Mode (provided that *Debug*[*DM*] = 0). • Interrupts are disabled. • TLB Refill exceptions will use the general exception vector instead of the TLB Refill vectors. • EPC will not be updated if another exception is taken.	R/W	Undefined
IE	0	Interrupt Enable. Functions as the master enable for software and hardware interrupts. 0: Interrupts disabled 1: Interrupts enabled	R/W	Undefined
0	26, 24, 18:16	These bits must be written as zero; returns zero on read.	0	0

[a] This bit it read-only and hardwired to 0 when the MMU is FMT- based.

6.12 Cause Register (CP0 Register 13, Select 0)

The *Cause* register contains information about the most recent processor exception. It also contains a field which controls software interrupt requests, and a bit that selects the exception vector used by the exception handler.

Figure 6-13 shows the format of the *Cause* register; Table 6-17 describes the *Cause* register fields.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BD	0	CE		0				IV	WP	0						IP7:IP0								0	ExcCode					0	

Figure 6-13 Cause Register

Table 6-17 Cause Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
BD	31	Branch Delay. Indicates whether the last exception taken occurred in a branch delay slot. 0: Not in delay slot 1: In delay slot	R	Undefined
CE	29:28	Coprocessor Exception. Contains the coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This field is written by hardware on every exception, but is undefined for all exceptions except the Coprocessor Unusable exception.	R	Undefined
IV	23	Interrupt Vector. Indicates whether an interrupt exception uses the general exception vector or the special interrupt vector. 0: Use the general exception vector (0x180) 1: Use the special interrupt vector (0x200)	R/W	Undefined
WP	22	Watch Postponed. Indicates that a Watch exception was deferred because Status$_{EXL}$ or Status$_{ERL}$ was set to a one when the Watch exception was detected. When this bit is set to one, the Watch exception is taken if Status$_{EXL}$ and Status$_{ERL}$ are both zero; thus software must clear this bit as part of the Watch exception handler to prevent a Watch exception loop.	R/W	Undefined
IP[7:2]	15:10	Interrupt Pending. Indicates an external interrupt is pending. 15: Hardware interrupt 5[a] 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	Interrupt Pending. Controls the request for software interrupts. 9: Request software interrupt 1 8: Request software interrupt 0	R/W	Undefined
ExcCode	6:2	Exception Code. Refer to Table 6-18.	R	Undefined

Table 6-17 Cause Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
0	30, 27:24, 21:16, 7, 1:0	Must be written as zero; returns zero on read.	0	0

a. Interrupt 5 may be used for the Timer interrupt, with external connection of the *SI_TimerInt* output to this input.

Table 6-18 Cause Register ExcCode Field

ExcCode Value	Mnemonic	Description
0	Int	Interrupt
1	Mod	TLB modification exception
2	TLBL	TLB exception (load or instruction fetch)
3	TLBS	TLB exception (store)
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 for load or store)
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved Instruction exception
11	CpU	Coprocessor Unusable exception
12	Ov	Arithmetic Overflow exception
13	Tr	Trap exception
14	-	Reserved
15	FPE	Floating-point exception
16-17	-	Available for an attached COP2
18	C2E	Coprocessor 2 exception
19-21	-	Reserved
22	MDMX	MDMX Unusable exception
23	WATCH	Reference to *WatchHi*/*WatchLo* address
24	MCheck	Machine Check exception
25-29	-	Reserved
30	CacheErr	Cache Error (Note that when there is a cache error in Debug Mode, this ExcCode is not used for the *Cause* register, but only for the *Debug* register.)
25-31	-	Reserved

6.13 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.

For synchronous (precise) exceptions, *EPC* contains either:

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

The processor does not write to the *EPC* register when the *EXL* bit in the *Status* register is set to 1.

Figure 6-14 shows the format of the *EPC* register; Table 6-19 describes the *EPC* register fields.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
EPC																															

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

Figure 6-14 EPC Register

Table 6-19 EPC Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	**Bits**			
EPC	63:0	Exception Program Counter	R/W	Undefined

6.14 Processor Identification (CP0 Register 15, Select 0)

The *Processor Identification (PRId)* register is a 32-bit, read-only register containing information that identifies the manufacturer, manufacturer options, processor identification, and revision level of the processor. Figure 6-15 shows the format of the *PRId* register; Table 6-20 describes the *PRId* register fields.

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

Figure 6-15 PRId Register

Table 6-20 PRId Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	**Bits**			
Company Options	31:24	Available to the CPU core user for company-dependent options. The value in this field is a direct input to the CPU core signal *SI_PRIdOpt*[7:0]. The default value of this field is zero.	R	Externally set

Table 6-20 PRId Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
Company ID	23:16	Identifies the company that designed or manufactured the processor. The value 1 in this field indicates that the processor is designed by MIPS Technologies, Inc.	R	1
Processor ID	15:8	Identifies the type of processor. This field allows software to distinguish between the various types of MIPS processors. For the 5K microprocessor core, this field contain the value 129 (binary 1000 0001).	R	129
Revision	7:0	Specifies the revision number of the processor. This field allows software to distinguish between different revisions of the same processor type. A value of 1 in this field indicates the first revision of the processor.	R	1

6.15 Configuration Register (CP0 Register 16, Select 0)

The *Config* register specifies various configuration options and capabilities of the 5K microprocessor core. Most of the fields in the *Config* register are initialized by hardware during the Reset exception or have constant values.

Figure 6-16 shows the format of the *Config* register; Table 6-21 describes the *Config* register fields.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	K23			KU			0			SB	ISD	0	0	DID	BM	BE	AT		AR			MT			0				K0		

Figure 6-16 Config Register

Table 6-21 Config Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
M	31	This bit is set to 1 to indicate that the *Config1* register is implemented.	R	1
K23	30:28	Specifies the kseg2 and kseg3 cache coherency algorithm to be used with an FMT-based MMU. Refer to Table 6-6 for the encoding of this field.	R/W	Undefined
KU	27:25	Specifies useg/kuseg cache coherency algorithm to be used with an FMT-based MMU. Refer to Table 6-6 for the encoding of this field.	R/W	Undefined
SB	21	Indicates whether SimpleBE bus mode is enabled. Set via the SI_SimpleBE[0] input pin: 0: No reserved byte enables on the EC interface 1: Only simple byte enables allowed on the EC interface	R	Externally set
ISD	20	Instruction Scheduling Disable. Disable the instruction scheduling feature of the processor. 0: Instruction Scheduling enabled 1: Instruction Scheduling disabled	R/W	0

Table 6-21 Config Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
DID	17	Dual Issue Disable 0: Dual issue enabled 1: Dual issue disabled	R/W	0
BM	16	Burst Mode. Indicates bus burst ordering. 0: Incremental 1: Interleaved	R	Externally set
BE	15	Indicates the current endian byte-ordering convention. 0: Little endian 1: Big endian	R	Externally set
AT	14:13	Architecture Type. Indicates the architecture type implemented by the processor. 1: MIPS64 with 32-bit address only 2: MIPS64 with 32/64-bit addresses The value 1 is used when the MMU type is FMT and the value 2 is used when the MMU type is TLB.	R	Build option[b]
AR	12:10	Architecture Revision. Specifies the architecture revision level. 0: Revision 1	R	0
MT	9:7	MMU Type. Specifies the type of MMU implemented. 1: Standard TLB 3: Standard FMT	R	Build option[a]
K0	2:0	Specifies the kseg0 cache coherency algorithm. Refer to Table 6-6 on page 107 for the encoding of this field.	R/W	2 (uncached)
0	24:22, 19:18, 6:3	Must be written as zero; returns zero on read.	0	0

[a] The setting of this field is determined when the 5K core is built with either a TLB or an FMT-based MMU.

6.16 Configuration Register 1 (CP0 Register 16, Select 1)

The *Config1* register is an adjunct to the *Config* register and is used to encode information about additional processor capabilities.

The I-cache and D-cache configuration parameters in this register include encoding of the number of sets per way, the line size, and the associativity. The cache size is equal to:

- Associativity * Line Size * Sets Per Way

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

Figure 6-17 shows the format of the *Config1* register; Table 6-22 describes the *Config1* register fields.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td colspan="6">MMU Size - 1</td><td colspan="3">IS</td><td colspan="3">IL</td><td colspan="3">IA</td><td colspan="3">DS</td><td colspan="3">DL</td><td colspan="3">DA</td><td>C2</td><td>MD</td><td>PC</td><td>WR</td><td>CA</td><td>EP</td><td>FP</td></tr>
</table>

Figure 6-17 Config1 Register

Table 6-22 Config1 Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
MMU Size - 1	30:25	Number of entries in the TLB minus one. This field can have the following values: 0: Value used when the MMU is not TLB-based 15: 16-entry TLB MMU (dual entries) 31: 32-entry TLB MMU (dual entries) 47: 48-entry TLB MMU (dual entries) All other values are reserved.	R	Build option[a]
IS	24:22	I-cache sets per way. 1: 128 sets 2: 256 sets 3: 512 sets All other values are reserved.	R	Preset[b]
IL	21:19	I-cache line size. 0: No I-cache present 4: 32 bytes All other values are reserved.	R	Preset[c]
IA	18:16	I-cache set associativity. 0: Direct mapped 1: 2-way 2: 3-way 3: 4-way All other values are reserved.	R	Preset[c]
DS	15:13	D-cache sets per way. 1: 128 sets 2: 256 sets 3: 512 sets All other values are reserved.	R	Preset[c]
DL	12:10	D-cache line size. 0: No D-cache present 4: 32 bytes All other values are reserved.	R	Preset[c]
DA	9:7	D-cache set associativity. 0: Direct mapped 1: 2-way 2: 3-way 3: 4-way All other values are reserved.	R	Preset[c]

Table 6-22 Config1 Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
C2	6	Coprocessor 2 implemented. 0: No coprocessor 2 implemented. 1: Coprocessor 2 implemented	R	Preset[c]
MD	5	MDMX ASE implemented. 0: No MDMX ASE implemented 1: MDMX ASE implemented	R	Preset[c]
PC	4	Performance Counter. This bit is 1 to indicate that the processor implements *Performance Counter* registers. 1: At least one performance counter register implemented	R	1
WR	3	*Watch* registers implemented. 1: At least one watch register implemented	R	1
CA	2	Code compression (MIPS16™ ASE) implemented. 0: No code compression	R	0
EP	1	EJTAG implemented. 1: EJTAG implemented	R	1
FP	0	FPU implemented. 0: No FPU 1: FPU implemented	R	Preset[c]
0	31, 6:5	Must be written as zero; returns zero on read.	0	0

[a] The setting of this field is determined when the CPU core is built with either a TLB- or an FMT-based MMU.

[b] The reset value of this field reflects the implemented hardware.

6.17 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, these registers duplicate some functions of the EJTAG debug solution.

Watch exceptions are taken only if the *Status* register's *EXL* and *ERL* bits are both zero. If either bit is a one, the *Cause* register's *WP* bit is set, and the Watch exception is deferred until both *EXL* and *ERL* bits are zero.

The CPU provides one *WatchLo*/*WatchHi* register pair, and ignores the select field of the MTC0/MFC0 and DMTC0/DMFC0 instructions for these registers. Software may determine that one pair of *WatchLo* and *WatchHi* registers are implemented via the *WR* bit of the *Config1* register and the *M* bit in the *WatchHi* register.

The *WatchLo* register specifies the virtual base address and the type of reference (instruction fetch, load, or store) to match. Figure 6-18 shows the format of the *WatchLo* register; Table 6-23 describes the *WatchLo* register fields.

<table>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="32">VAddr</td></tr>
</table>

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="29">VAddr</td><td>I</td><td>R</td><td>W</td></tr>
</table>

Figure 6-18 WatchLo Register

Table 6-23 WatchLo Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
VAddr	63:3	Virtual Address. This field specifies the virtual address to match. Note that this is a doubleword address, because bits 2:0 are used to control the type of match. Bits 2:0 are ignored for the watchpoint address comparison.	R/W	Undefined
I	2	Instruction. If this bit is one, Watch exceptions are enabled for instruction fetches that match the address.	R/W	0
R	1	Read. If this bit is one, Watch exceptions are enabled for loads that match the address.	R/W	0
W	0	Write. If this bit is one, Watch exceptions are enabled for stores that match the address.	R/W	0

6.18 WatchHi Register (CP0 Register 19)

The *WatchHi* register contains information that qualifies the virtual address specified in the *WatchLo* register, including an *ASID*, a *G* (Global) bit, and an optional address mask. If the *G* bit is one, any virtual address reference that matches the specified address will cause a Watch exception. If the *G* bit is zero Watch exception is caused only by those virtual address references for which the ASID value in the *WatchHi* register matches the ASID value in the *EntryHi* register. The optional *Mask* field provides address masking that qualifies the address specified in *WatchLo*.

Figure 6-19 shows the format of the *WatchHi* register; Table 6-24 describes the *WatchHi* register fields.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>M</td><td>G</td><td colspan="6">0</td><td colspan="8">ASID</td><td colspan="4">0</td><td colspan="9">MASK</td><td colspan="3">0</td></tr>
</table>

Figure 6-19 WatchHi Register

Table 6-24 WatchHi Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
M	31	This bit is set to zero to indicate that only one pair of *WatchHi*/*WatchLo* registers are implemented.	R	0

Table 6-24 WatchHi Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
G	30	Global. If this bit is one, any address that matches the address specified in the *WatchLo* register will cause 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
ASID	23:16	ASID. Specifies the ASID value required to match the ASID value in the *EntryHi* register, when the *G* bit in the *WatchHi* register is set to zero.	R/W	Undefined
Mask	11:3	Optional bit mask that qualifies the address in the *WatchLo* register. Any bit in this field that is set prevents the corresponding address bit from participating in the address match.	R/W	Undefined
0	29:24, 15:12, 2:0	Must be written as zero; returns zero on read.	0	0

6.19 XContext Register (CP0 Register 20, Select 0)

Like the *Context* register, the *XContext* register contains a pointer to an entry in the page table entry (PTE) array, an operating system data structure that stores virtual-to-physical address translations. When there is a TLB exception, the operating system software loads the TLB with the missing translation from the PTE array. The *XContext* register provides a different view of the information provided in the *BadVAddr* register, and puts it in a form more directly usable by TLB exception handlers. The *XContext* register is for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space. The operating system sets this register's *PTEBase* field, as needed. Normally, the operating system uses the X*Context* register to address the current page map, which resides in the kernel-mapped segment kseg3.

On a 64-bit TLB miss, the *R* field contains bits 63:62 of the virtual address, which select the mapped region. The 27-bit *BadVPN2* field contains bits 39:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format may be used directly to address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

The *R* and *BadVPN2* fields of the *XContext* register are updated on TLB exceptions and undefined following Address Error exceptions.

Figure 6-20 shows the format of the *XContext* register; Figure 6-20 describes the *XContext* register fields.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
PTEBase																															R

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	BadVPN2																											0			

Figure 6-20 XContext Register Format

Table 6-25 XContext Register Fields

Field		Description	Read/Write	Reset State
Name	Bits			
PTEBase	63:33	The Page Table Entry Base. This field is normally written with a value that allows the operating system to use the *XContext* register as a pointer into the current PTE array in memory.	R/W	Undefined
R	32:31	Region. Contains bits 63:62 of the virtual address in the *BadVAddr* register, which select the address space as follows: 00_2 = xuseg 01_2 = xsseg 10_2 = Reserved 11_2 = xkseg	R	Undefined
BadVPN2	30:4	This field contains bits 39:13 of the virtual address in the *BadVAddr* register.	R	Undefined
0	3:0	Must be written as zeroes; returns zeroes when read.	R	0

6.20 Debug Register (CP0 Register 23, Select 0)

The *Debug* register describes the cause of the most recent debug exception or exception that occurred in Debug Mode. It also provides status information for various machine resources available in Debug Mode. In addition, it enables the single step exception (used only in Non-Debug Mode).

When this register is read in Non-Debug Mode, only the *DM* bit and *EJTAGver* field are valid; the value of all other bits and fields are **UNPREDICTABLE**. In Non-Debug Mode, writes to the Debug register are ignored.

Some of the bits and fields are only updated on debug exceptions and/or exceptions in Debug Mode:

- *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB*, *DINT* and *DDBLImpr* are updated on both debug exceptions and on exceptions in Debug Mode.
- *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 are undefined after an exception in Debug Mode.
- *DBD* is updated on both debug exceptions and on exceptions in Debug Mode.

Figure 6-21 shows the format of the *Debug* register; Table 6-26 describes the register fields.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DBD	DM	NoDCR	LSNM	Doze	Halt	CountDM	0	MCheckP	CacheEP	DBusEP	IEX1	0	DDBLimpr	EJTAGver	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EJTAGver	DExcCode					0	SSt	0		DINT	DIB	DDBS	DDBL	DBp	DSS

Figure 6-21 Debug Register

Table 6-26 Debug Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	**Bits**			
DBD	31	Debug Branch Delay. Indicates whether the last debug exception or exception in Debug Mode occurred in a branch delay slot. 0: Exception not in delay slot 1: Exception in delay slot	R	Undefined
DM	30	Debug Mode. 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
NoDCR	29	Indicates that the dseg memory segment is present. 0: dseg is present	R	0
LSNM	28	Load Store Normal Memory. Controls load/store accesses to dseg and non-dseg memory. 0: Access references dseg 1: Access references non-dseg memory	R/W	0
Doze	27	Indicates if the processor was in low-power mode when a debug exception occurred. 0: Processor not in low-power mode 1: Processor in low-power mode	R	Undefined
Halt	26	Indicates that the internal system bus clock was stopped when the debug exception occurred. 0: Internal system bus clock running 1: Internal system bus clock stopped	R	Undefined
CountDM	25	Count Debug Mode. This bit is set to 1 to indicate that the *Count* register always counts.	R	1
MCheckP	23	Machine Check Exception Pending. Set when a Machine Check exception is pending, either because a Machine Check exception has been signaled by hardware, or software has set this bit to 1. This bit is cleared when the processor takes the Machine Check exception. Note that if software writes a 1 to this bit, the processor will take a Machine Check exception as soon as the exception is no longer masked by higher priority exceptions or by $Debug_{IEXI}$. This bit allows Machine Check exceptions caused by Non-Debug Mode software, but first signaled when the processor is in Debug Mode, to be deferred until after the execution of the DERET instruction.	R/W1	0

Table 6-26 Debug Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
CacheEP	22	Cache Error Exception Pending. Set when a Cache Error exception is pending, either because a Cache Error exception has been signaled by hardware, or because software has set this bit to 1. This bit is cleared when the processor takes the Cache Error exception. Note that if software writes a 1 to this bit, the processor will take a Cache Error exception as soon as the exception is no longer masked by higher priority exceptions or by $Debug_{IEXI}$. This bit allows Cache Error exceptions caused by Non-Debug Mode software, but first signaled when the processor is in Debug Mode, to be deferred until after the execution of the DERET instruction.	R/W1	0
DBusEP	21	Data Bus Error Exception Pending. Set when a Data Bus Error exception is pending, either because a Data Bus Error exception has been signaled by hardware, or because software has set this bit to 1. This bit is cleared when the processor takes the Data Bus Error exception. Note that if software writes a 1 to this bit, the processor will take a Data Bus Error exception as soon as the exception is no longer masked by higher priority exceptions or by $Debug_{IEXI}$. This bit allow Data Bus Error exceptions caused by Non-Debug Mode software, but first signaled when the processor is in Debug Mode, to be deferred until after the execution of the DERET instruction. In Debug Mode, a Data Bus Error exception applies to a Debug Mode Data Bus Error exception.	R/W1	0
IEXI	20	Imprecise Error Exception Inhibit. Set when the processor takes a debug exception, or when Debug Mode is re-entered. This bit is cleared by execution of the DERET instruction and modifiable by software. When *IEXI* is set, the Bus Error, Cache Error, and Machine Check exceptions are deferred until this bit is cleared.	R/W	0
DDBLImpr	18	Debug Data Break Imprecise. Indicates that an Imprecise Debug Data Break was the cause of the debug exception, or that an Imprecise Debug Data Break was signaled after another debug exception occurred. This bit is cleared on exception in Debug Mode. 0: No imprecise data break indication, or exception in Debug Mode occurred 1: Imprecise data break indication	R	Undefined
EJTAGver	17:15	Indicates the EJTAG version implemented: 0: Version 1 and 2.0 1: Version 2.5 2: Version 2.6 3-7: Reserved	R	2
DExcCode	14:10	Debug Exception Code. Indicates the cause of the latest exception in Debug Mode. This field is encoded as the *ExcCode* field in the *Cause* register for use by software in servicing non-debug exceptions that occur in Debug Mode. Value is undefined after a debug exception.	R	Undefined

Table 6-26 Debug Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
SSt	8	Debug Single Step. Enables Debug single step exception. 0: Single step not enabled 1: Single step enabled	R/W	0
DINT	5	Debug Interrupt. Indicates that a Debug Interrupt exception occurred. This bit is cleared on exception in Debug Mode. 0: No Debug Interrupt exception 1: Debug Interrupt exception	R	Undefined
DIB	4	Debug Instruction Break. Indicates that a Debug Instruction Break exception occurred. This bit is cleared on exception in Debug Mode. 0: No Debug Instruction Break exception 1: Debug Instruction Break exception	R	Undefined
DDBS	3	Debug Data Break Store. Indicates that a Debug Data Break exception occurred on a store. This bit is cleared on exception in Debug Mode. 0: No Debug Data Break exception on a store 1: Debug Date Break exception on a store	R	Undefined
DDBL	2	Debug Data Break Load. Indicates that a Debug Data Break exception occurred on a load. This bit is cleared on exception in Debug Mode. 0: No Debug Data Break exception on a load 1: Debug Data Break exception on a load	R	Undefined
DBp	1	Debug Breakpoint. Indicates that a debug software Breakpoint exception occurred. This bit is cleared on exception in Debug Mode. 0: No Debug software Breakpoint exception 1: Debug software Breakpoint exception	R	Undefined
DSS	0	Debug Single Step. Indicates that a Debug Single step exception occurred. This bit is cleared on exception in Debug Mode. 0: No Debug Single Step exception 1: Debug Single Step exception	R	Undefined
0	24, 19, 7:6	Must be written as zero; returns zero on read.	0	0

6.21 Debug Exception Program Counter Register (CP0 Register 24, Select 0)

The *Debug Exception Program Counter* (*DEPC*) register is a 64-bit read/write register that contains the address at which execution resumes after servicing a debug exception or an exception in Debug Mode.

This register is updated by hardware when a debug exception occurs, and when any exception occurs in Debug Mode. This register contains two possible values:

1. The virtual address of the instruction that was the direct cause of the debug exception or the exception in Debug Mode or,
2. The virtual address of the immediately preceding branch or jump instruction, when the instruction causing the exception is in a branch delay slot, and the *Debug Branch Delay* (*BDB*) bit in the *Debug* register is set.

Figure 6-22 shows the format of the *DEPC* register; Table 6-27 describes the *DEPC* register fields.

<table>
<tr><th>63</th><th>62</th><th>61</th><th>60</th><th>59</th><th>58</th><th>57</th><th>56</th><th>55</th><th>54</th><th>53</th><th>52</th><th>51</th><th>50</th><th>49</th><th>48</th><th>47</th><th>46</th><th>45</th><th>44</th><th>43</th><th>42</th><th>41</th><th>40</th><th>39</th><th>38</th><th>37</th><th>36</th><th>35</th><th>34</th><th>33</th><th>32</th></tr>
<tr><td colspan="32">DEPC</td></tr>
</table>

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="32">DEPC</td></tr>
</table>

Figure 6-22 DEPC Register

Table 6-27 DEPC Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	**Bits**			
DEPC	63:0	Debug Exception Program Counter	R/W	Undefined

6.22 Performance Counter Register (CP0 Register 25, select 0-3)

The 5K processor defines two performance counters and two associated control registers, which are mapped to CP0 register 25. The select field of the MTC0/MFC0 instructions are used to select the specific register accessed by the instruction, as shown in Table 6-28.

Table 6-28 Performance Counter Register Selects

Select[1:0]	Register
0	Register 0 Control
1	Register 0 Count
2	Register 1 Control
3	Register 1 Count

Each counter is a 32-bit read/write register and is incremented by one each time the countable event, specified in its associated control register, occurs. Each counter can independently count one type of event at a time.

Bit 31 of each of the two counters are ANDed with an interrupt enable bit, *IE*, of their respective control register, and then ORed together with hardware interrupt 5 input to generate an interrupt on counter overflow. Counting is not affected by the interrupt indication. This output is cleared when the counter wraps to zero, and may be cleared in software by writing a value with bit 31 = 0 to the *Performance Counter Count* registers.

Figure 6-23 shows the format of the *Performance Counter Control* register; Table 6-29 describes the *Performance Counter Control* register fields.

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

Figure 6-23 Performance Counter Control Register

Table 6-29 Performance Counter Control Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
M	31	If this bit is one, another pair of *Performance Control* and *Counter* registers is implemented at a MTC0 or MFC0 select field value of ‘n+2’ and ‘n+3’.	R	1 for performance counter 0 0 for performance counter 1
Event	8:5	Counter event enabled for this counter. Possible events are listed in Table 6-30.	R/W	Undefined
IE	4	Counter Interrupt Enable. This bit masks bit 31 of the associated count register from the interrupt exception request output.	R/W	0
U	3	Count in User Mode. When this bit is set, the specified event is counted in User Mode.	R/W	Undefined
S	2	Count in Supervisor Mode. When this bit is set, the specified event is counted in Supervisor Mode.	R/W	Undefined
K	1	Count in Kernel Mode. When this bit is set, count the event in Kernel Mode when *EXL* and *ERL* both are 0.	R/W	Undefined
EXL	0	Count when *EXL*. When this bit is set, count the event when *EXL* = 1 and *ERL* = 0.	R/W	Undefined
0	30:9, 2	Must be written as zeroes; returns zeroes when read.	0	0

Table 6-30 describes the events countable with the two performance counters. The operation of a counter is **UNPREDICTABLE** for events which are specified as Reserved.

Table 6-30 Performance Counter Count Register Field Descriptions

Event Number	Counter 0	Counter 1
0	Cycles	Cycles
1	Instructions fetched	Instructions executed
2	Load/pref(x)/sync/cache-ops executed	Load/pref(x)/sync/cache-ops executed
3	Stores (including conditional stores) executed	Stores (including conditional stores) executed
4	Conditional stores executed	Conditional stores executed
5	Failed conditional stores	Floating-point instructions executed
6	Branches executed	Data cache line evicted
7	ITLB miss	TLB miss exceptions
8	DTLB miss	Branch mispredicted
9	Instruction cache miss	Data cache miss

Table 6-30 Performance Counter Count Register Field Descriptions

Event Number	Counter 0	Counter 1
10	Instruction scheduled	Instruction stall in M stage due to scheduling conflicts
11	Reserved	Reserved
12	Reserved	Reserved
13	Reserved	Reserved
14	Dual issued instructions executed	Reserved
15	Instructions executed	COP2 instructions executed

Event 0, counter 0 & 1: Cycles

The counter is incremented by one on each clock cycle.

Event 1, counter 0: Instructions fetched

The counter is incremented by the number of instructions (0, 1, or 2) fetched by the instruction buffer in the previous cycle.

Event 1, counter 1: Instructions executed

The counter is incremented by the number of instructions (0, 1 or 2) which have completed their execution in the integer unit or the floating-point unit in the previous cycle. With respect to the performance counters, an instruction has completed its execution if it has passed its M stage without being killed or if the instruction was a SYSCALL, BREAK, SDBBP, or trap instruction caused the corresponding exception. Note that a load instruction is thus regarded as executed if it has completed the M stage, even though the load may have been scheduled, and therefore its data has not yet been written to a GPR. MDU and arithmetic coprocessor instructions are also counted when they have completed the M stage, regardless of how many additional cycles they will require to complete execution.

Event 2, counter 0 &1: Load/pref(x)/sync/cache-ops executed

The counter is incremented by one each time a load, pref(x), sync, or cache instruction has been executed.

Event 3, counter 0 & 1: Stores (including conditional stores) executed

The counter is incremented by one each time a store instruction has been executed. A store instruction is considered executed when it has completed its M stage without being killed, even when it has not yet written the uncached data to external memory. Note that a store conditional is considered executed even if it fails to perform the store because the LL bit has been cleared.

Event 4, counter 0 & 1: Conditional stores executed

Similar to event 3, counter 1, but only affected by store conditional instructions.

Event 5, counter 0: Failed conditional stores

The counter is incremented by one each time a store conditional instruction fails the store.

Event 5, counter 1: Floating-point instructions executed

The counter is incremented by one each time a floating-point instruction has been executed.

Event 6, counter 0: Branches executed

The counter is incremented by one each time a conditional branch instruction has been executed.

Event 6, counter 1: Data cache line evicted

The counter is incremented by one each time a line is evicted from the data cache.

Event 7, counter 0: ITLB miss

The counter is incremented by one each time there is a miss in the ITLB.

Event 7, counter 1: TLB miss exceptions

The counter is incremented by one each time a TLB miss exception is taken.

Event 8, counter 0: DTLB miss

The counter is incremented by one each time there is a miss in the DTLB.

Event 8, counter 1: Branch mispredicted

The counter is incremented by one each time a conditional branch is mispredicted.

Event 9, counter 0: Instruction cache miss

The counter is incremented by one each time there is a miss in the instruction cache.

Event 9, counter 1: Data cache miss

The counter is incremented by one each time there is a miss in the data cache.

Event 10, counter 0: Instructions scheduled

The counter is incremented by one each time an instruction has been scheduled.

Event 10, counter 1: Instruction stall in M stage due to scheduling conflicts

The counter is incremented by one for each clock cycle when an instruction causes an M stage pipeline stall due to scheduling conflicts.

Event 14, counter 0: Dual issued instructions executed

The counter is incremented by two each time an instruction pair that was dual issued has been executed. See Section , "Event 1, counter 1: Instructions executed" for a description of when an instruction is considered executed.

Event 15, counter 0: Instructions executed

Identical to event 1, counter 1.

Event 15, counter 1: COP2 instructions executed

The counter is incremented by one each time a COP2 instruction has been executed.

Figure 6-24 shows the format of the *Performance Counter Count* register; Table 6-31 describes the *Performance Counter Count* register fields.

The performance counter resets to a low-power state, in which none of the counters will start counting events until software has enabled event counting, using an MTC0 instruction to the Performance Counter Control Registers.

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="32">Counter</td></tr>
</table>

Figure 6-24 Performance Counter Count Register

Table 6-31 Performance Counter Count Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
Counter	31:0	Counter	R/W	Undefined

6.23 ErrCtl Register (CP0 Register 26, Select 0)

The *ErrCtl* register controls parity protection of data and instruction caches and provides for software testing of the way-selection RAM.

Parity protection can be enabled or disabled using the *PE* bit. When parity is enabled, the *PO* bit controls overwrite of calculated parity for the CACHE instructions Indexed Store operation. A CACHE Index Load Tag operation will always cause this register to be updated with the parity bits read, along with the update of the *DataLo* and *DataHi* register when parity is enabled. When parity is disabled, the contents of the *P* field is **UNPREDICTABLE.**

The way- selection RAM test mode is enabled by setting the *WST* bit. This mode is intended for software testing of the way-selection RAM and data RAM. It modifies the functionality of the CACHE Index Load Tag and Index Store Tag operations so that they modify the way-selection RAM but do not modify the TAG RAMs. In this mode, the lower 14 bits of the *PTagLo* field of the *TagLo* register are used as the source and destination for load and store operations for the way-selection RAM. Refer to Figure 8-2 for the layout of the way-selection RAM. The WS bits, Dirty bits and Dirty Parity bits (optional) are accessible through *PTagLo[12:7]*, *PTagLo[7:4]*, and *PTagLo[3:0]*, respectively. In addition, when the WST bit is set, the CACHE Index Store Data can be used for testing the data RAM. When the *WST* bit is set, the CACHE Index Store Tag is used to the write to the way-selection RAM rather than the tag RAM and these writes are performed with parity overwrite, disregarding the setting of the *PO* bit.

Figure 6-25 shows the format of the *ErrCtl* register; Table 6-32 describes the *ErrCtl* register fields.

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

Figure 6-25 ErrCtl Register

Table 6-32 ErrCtl Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
PE	31	Parity Enable. This bit enables or disables the cache parity protection. 0: Parity disabled 1: Parity enabled	R/W	0
PO	30	Parity Overwrite. If set, the contents of the *P* field overwrites calculated parity when data is written to the cache for the CACHE instructions indexed operations. 0: Use calculated parity 1: Use bits in P field for parity	R/W	0

Table 6-32 ErrCtl Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
WST	29	Way Selection Test. If set, way-selection RAM test mode is enabled. 0: Test mode disabled 1: Test mode enabled	R/W	0
P	7:0	Parity bits read from or written to a cache data RAM. P[0] is even parity for the least-significant byte of the requested data.	R/W	Undefined
0	28:8	Must be written as zeroes; returns zeroes when read.	0	0

6.24 CacheErr Register (CP0 Register 27, Select 0)

The *CacheErr* register provides an interface with the cache error-detection logic. When a Cache Error exception is signaled, the fields of this register are set accordingly.

Figure 6-26 shows the format of the *CacheErr* register; Table 6-33 describes the *CacheErr* register fields.

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>ER</td><td>0</td><td>ED</td><td>ET</td><td colspan="2">0</td><td>EB</td><td>EF</td><td>0</td><td>EW</td><td colspan="6">0</td><td colspan="16">Index</td></tr>
</table>

Figure 6-26 CacheErr Register

Table 6-33 CacheErr Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
ER	31	Error Reference. Indicates the type of reference that encountered an error. 0: Instruction 1: Data	R	Undefined
ED	29	Error Data. Indicates a data RAM error. 0: No data RAM error detected 1: Data RAM error detected	R	Undefined
ET	28	Error Tag. Indicates a tag RAM error. 0: No tag RAM error detected 1: Tag RAM error detected	R	Undefined
EB	25	Error Both. Indicates that a data cache error occurred in addition to an instruction cache error. 0: No additional data cache error 1: Additional data cache error In the case of an additional data cache error, the remainder of the bits in this register are set according to the instruction cache error.	R	Undefined

Table 6-33 CacheErr Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
EF	24	Error Fatal. Indicates that a fatal cache error has occurred. There are a few situations where software will not be able to get all information about a cache error from the *CacheErr* register. These situations are fatal because software cannot determine which memory locations have been affected by the error. To enable software to detect these cases, the *EF* bit (bit 24) has been added to the *CacheErr* register. The following 6 cases are indicated as fatal cache errors by the *EF* bit: 1. Dirty parity error in dirty victim (dirty bit cleared in tag) 2. Tag parity error in dirty victim 3. Data parity error in dirty victim 4. WB store miss and EW error at the requested index 5. Dual/Triple errors from different transactions, e.g. scheduled and non-scheduled load. 6. Multiple data cache errors detected before the first instruction of the cache error handler is issued. In addition to the above, simultaneous instruction and data cache errors as indicated by *CacheErr*[*EB*] will cause information about the data cache error to be unavailable. However, that situation is not indicated by *CacheErr*[*EF*].	R	Undefined
EW	22	Error Way. Indicates a way selection RAM error. 0: No way selection RAM error detected 1: Way selection RAM error detected	R	Undefined
Way	21:20	Way. Specifies the cache way in which the error was detected. It is not valid if a Tag RAM error is detected (ET=1).	R	Undefined
Index	15:0	Index. Specifies the cache index of the double word in which the error was detected. The way of the faulty cache is written by hardware in the *Way* field. Software must combine the *Way* and *Index* read in this register with cache configuration information in the *Config1* register in order to obtain an index which can be used in an indexed CACHE instruction to access the faulty cache data or tag. Note that *Index* is aligned as a byte index, so it does not need to be shifted by software before it is used in an indexed CACHE instruction. *Index* bits [4:3] are undefined upon tag RAM errors and *Index* bits above the MSB actually used for cache indexing will also be undefined.	R	Undefined
0	30, 27:26, 23, 21:16	Must be written as zeroes; returns zeroes when read.	0	0

6.25 TagLo Register (CP0 Register 28, Select 0)

The *TagLo* register is a read/write register that acts as the interface to the cache tag array for both the instruction and data caches. The Index Store Tag and Index Load Tag operations of the CACHE instruction use the *TagLo* register as the source or destination of tag information, respectively.

Figure 6-27 shows the format of the *TagLo* register; Table 6-34 describes the *TagLo* register fields.

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

Figure 6-27 TagLo Register

Table 6-34 TagLo Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	**Bits**			
PTagLo	31:8	Specifies the upper address bits for the cache tag. Bit 31 of this field corresponds to bit 35 of the physical address. Bit 8 corresponds to bit 12 of the physical address.	R/W	Undefined
PState	7:6	Specifies the state bits for the cache line. It can have the following values: 0: Invalid line. 2: Valid clean line. 3: Valid dirty line. The value 1 is not defined. If software sets the PState of a cache line to 1, the operation of the cache will be **UNDEFINED**.	R/W	Undefined
L	5	Lock. Specifies the state of the lock bit for the cache line. 0: The line is not locked. 1: The line is locked. When locked, a cache line will not be replaced by the cache subsystem. A cache line will be disabled (will not generate a hit or be replaced by a refill) if it is invalid and locked. It is only possible to obtain this state by using the CACHE Index Store Tag instruction.	R/W	Undefined
P	0	Parity. Specifies the parity bit for the cache tag. This bit is updated with tag parity on CACHE instruction Index Load Tag operations and used as tag parity on Index Store Tag operations when the *PO* bit of the *ErrCtl* register is set. The Index Store Tag operation uses computed parity when the *PO* bit of the *ErrCtl* register is not set.	R/W	Undefined
0	4:1	Must be written as zero; returns zero on read.	0	0

6.26 DataLo Register (CP0 Register 28, Select 1)

The *DataLo* and *DataHi* registers act as the interface to the cache data arrays in both the instruction and data caches, and are intended for diagnostic operations only. The Index Load Tag operation of the *CACHE* instruction reads the data field of the indexed way of the cache data RAM. (For the layout of the cache data RAM, refer to Figure 8-2.)

The *DataLo* register is a 64-bit register that holds all the data read from the data field. The *DataHi* register holds the upper 32 bits of the same data.

The *DataLo* and *DataHi* registers can be both read and written by software. Special restrictions apply to a program wishing to write the *DataLo* and *DataHi* registers using the MTC0 instruction rather than the DMTC0 instruction. Because the *DataHi* register provides access to the upper 32 bits of the *DataLo* register, a program should always do a MTC0 to the *DataLo* register BEFORE a MTC0 to the *DataHi* register. No special restrictions apply to MFC0 from these registers. A program which accesses these registers using the DMTC0/DMFC0 instructions need only access the *DataLo* register.

Figure 6-28 shows the format of the *DataLo* register; Table 6-35 describes the *DataLo* register fields.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Data																															

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

Figure 6-28 DataLo Register

Table 6-35 DataLo Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
Data	63:0	Data read from the data array of the cache.	R/W	Undefined

6.27 TagHi Register (CP0 Register 29, Select 0)

The *TagHi* register is not used in the 5K processor core.

Figure 6-29 shows the format of the *TagHi* register; Table 6-36 describes the *TagHi* register fields.

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

Figure 6-29 TagHi Register

Table 6-36 TagHi Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:0	Must be written as zero; returns zero on read.	0	0

6.28 DataHi Register (CP0 Register 29, Select 1)

This register is described in the section Section 6.26, "DataLo Register (CP0 Register 28, Select 1)" Figure 6-30 shows the format of the *DataHi* register; Table 6-37 describes the *DataHi* register fields.

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="32">Data</td></tr>
</table>

Figure 6-30 DataHi Register

Table 6-37 DataHi Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	**Bits**			
Data	31:0	High-order data read from the cache data array.	R/W	Undefined

6.29 ErrorEPC (CP0 Register 30, Select 0)

The *ErrorEPC* register is a read/write register, similar to the *EPC* register, except that *ErrorEPC* is used for error exceptions. All bits of the *ErrorEPC* register are significant. This register is also used to store the program counter on Reset, Soft Reset, and non-maskable interrupt (NMI) exceptions.

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

- the virtual address of the instruction that caused the exception
- the virtual address of the immediately preceding branch or jump instruction, when the instruction causing the exception is in a branch delay slot
- unrelated to the event that caused the exception

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

Figure 6-31 shows the format of the *ErrorEPC* register; Table 6-38 describes the *ErrorEPC* register fields.

<table>
<tr><th>63</th><th>62</th><th>61</th><th>60</th><th>59</th><th>58</th><th>57</th><th>56</th><th>55</th><th>54</th><th>53</th><th>52</th><th>51</th><th>50</th><th>49</th><th>48</th><th>47</th><th>46</th><th>45</th><th>44</th><th>43</th><th>42</th><th>41</th><th>40</th><th>39</th><th>38</th><th>37</th><th>36</th><th>35</th><th>34</th><th>33</th><th>32</th></tr>
<tr><td colspan="32">ErrorEPC</td></tr>
</table>

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="32">ErrorEPC</td></tr>
</table>

Figure 6-31 ErrorEPC Register

Table 6-38 ErrorEPC Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	**Bits**			
ErrorEPC	63:0	Error Exception Program Counter	R/W	Undefined

6.30 Debug Exception SAVE (DESAVE) (CP0 register 31)

This register is used by the debug exception handler to save one of the GPRs, which can then be used to save the rest of the context in a pre-determined memory location, for example, in the EJTAG Probe. This register ensures that exception handlers and other types of code can be safely debugged, even in situations where the existence of a valid stack for context saving cannot be assumed.

Figure 6-32 shows the format of the *DESAVE* register; Table 6-39 describes the *DESAVE* register fields.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
DESAVE																															

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

Figure 6-32 DESAVE Register

Table 6-39 DESAVE Register Fields

Fields		Description	Read/Write	Reset State
Name	**Bits**			
DESAVE	63:0	Simple Read/Write register	R/W	Undefined

Chapter 7

Hardware and Software Initialization

This chapter describes the hardware and software initialization of the MIPS64 5K processor core. The 5K core performs only a minimal amount of hardware initialization and relies on software to fully initialize the device.

This chapter contains the following sections:

- Section 7.1, "Hardware-Initialized Processor State"
- Section 7.2, "Software-Initialized Processor State"

7.1 Hardware-Initialized Processor State

The 5K processor core, like most MIPS processors, is not fully initialized by Reset. Only a minimal subset of the processor state is cleared, which is sufficient for the processor to begin executing 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.

7.1.1 Coprocessor 0 State

Much of the software-visible hardware initialization occurs in *Coprocessor 0* (*CP0*). The *CP0* registers and their reset values are documented in Chapter 6. The initialization performed by Reset and SoftReset is documented in Section 5.2, "Reset Exception" and Section 5.3, "Soft Reset Exception".

7.1.2 TLB Initialization

Each 5K 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 (“TLB Shutdown” occurs when two or more TLB entries match on a single address). This bit is not visible to software.

7.1.3 Bus State Machines

When a Reset or SoftReset exception is taken, all pending bus transactions are aborted, and the state machines in the Bus Interface Unit are reset.

7.1.4 Static Configuration Inputs

All static configuration inputs (for example, defining endianess) should only be changed during Reset.

7.1.5 Fetch Address

Upon Reset/SoftReset, unless the EJTAGBOOT option is used, the fetch is directed to virtual address 0xFFFFFFFFBFC00000 (physical address 0x01FC00000). This address is in kseg1, which is unmapped and uncached, so that the TLB and caches do not require hardware initialization. the EJTAGBOOT option is described in Section 10.5.1.3, "EJTAGBOOT and NORMALBOOT Instructions" on page 189.

7.2 Software-Initialized Processor State

This section describes the software required to initialize the COP0 registers, general-purpose registers, TLB, and caches.

7.2.1 Coprocessor 0 Registers

Miscellaneous CP0 state needs to be initialized by software before exiting the boot code:

- The following fields in the *Status* register must be initialized: *CU*, *RE*, *MX* (if a coprocessor implementing the MDMX ASE is attached to the processor), *PX*, *IM*, *KX*, *SX*, *UX*, *KSU*, *EXL*, and *IE*.
- The following fields of the *Cause* register must be initialized: IV, WP, and IP[1:0].
- If the MMU is built with the FMT option, the following fields of the *Config* register must be initialized: *K23* and *KU*.
- If timer interrupts are used, the *Count* and *Compare* registers must be set to a known value. The write to the *Compare* register will also clear any pending timer interrupts (and thus the *Count* register should be set before the *Compare* register, to avoid any unexpected interrupts).

7.2.2 Register File

On power-up, the register file is 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. However, the boot code can initialize the register file for added safety.

7.2.3 TLB

Because of the hidden bit indicating initialization, the 5K core does not require TLB initialization on Reset. However, this is an implementation-specific feature of the 5K core and cannot be relied upon when writing generic code for MIPS64 processors.

When initializing the TLB, care must be taken to avoid creating a “TLB Shutdown” condition, where two TLB entries match on a single address. To avoid this, unique virtual addresses must be written to each TLB entry.

7.2.4 Caches

On power-up, the cache tag and data arrays are in an unknown state and are not affected by Reset. Every tag in the cache arrays must be initialized to the invalid state by using the CACHE instruction (typically, the Index Invalidate cache operation) to clear it to zero. No cache line should be accessed by a cached access before it has been properly initialized. It is advantageous to initialize the instruction cache first, so that the initialization of the data cache can be done by code executing from the cache.

Chapter 8

Cache Organization and Operation

This chapter describes the organization and operation of the cache subsystem in the 5K microprocessor core. It contains the following sections.

8.1 Introduction

The 5K microprocessor core supports two caches—an instruction cache (I-cache) and a data cache (D-cache), each up to 64Kbytes in size. The use of separate caches allows instruction fetches and data accesses to occur at the same time. The caches are 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 to complete.

Cache refills are performed using a four-entry read buffer that performs four doubleword burst reads from memory, using incremental or sub-block refill ordering. While refills are in progress, the caches can continue processing hits. Streaming is also supported, in which instructions and data are forwarded during cache refills. In addition, the read buffer permits data cache-refills to proceed while a cache line is being written back to memory. Cache performance is further enhanced by special way-selection (WS) logic that implements a least-recently used (LRU) algorithm for way selection when a cache line is replaced.

To optimize performance, there is one read buffer and two, four-doubleword write buffers. One of the write buffers is used for writing back dirty cache lines to memory (called *evictions*), and the other is used for uncached stores, write-through cached stores, and for merging and gathering data for high-speed burst writes to memory.

Many cache characteristics can be configured by the user, including cache way size (4, 8, or 16 Kbytes) and set associativity (direct-mapped, 2-way, 3-way, or 4-way set associative). These parameters are independent for the two caches. The 5K supports a variety of different cache configurations (size, associativity, parity) including configurations without caches.

The 5K supports special instructions (CACHE, PREF, and PREFX) which can be used for testing, performance analysis, and code optimization. For 5K-based systems that require guaranteed deterministic behavior of certain pieces of code, cache lines can be locked using the CACHE instruction.

8.2 Cache Organization

Figure 8-1 shows the organization of the 5K cache subsystem. Each cache contains four components: the I-cache contains the instruction cache controller, the I-Tag RAM, I-Data RAM, and I-WS RAM; the D-cache contains the data cache controller, the D-Tag RAM, D-Data RAM, and D-WS RAM.

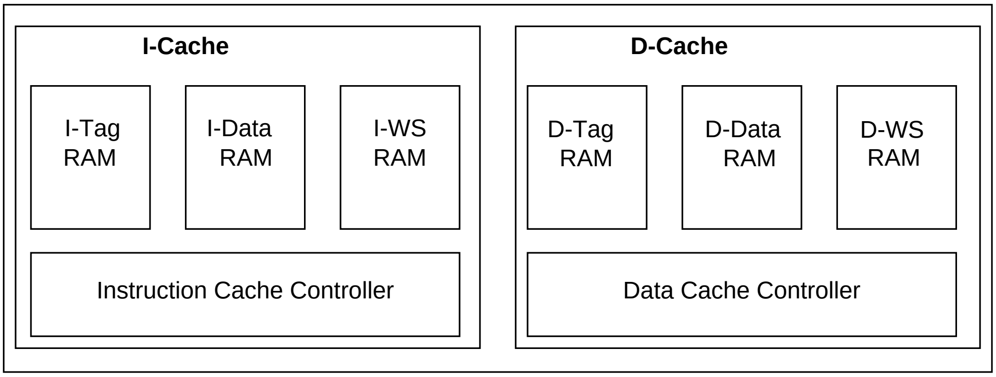


Figure 8-1 5K Cache Subsystem Organization

The contents of the individual RAM components are shown in Figure 8-2.

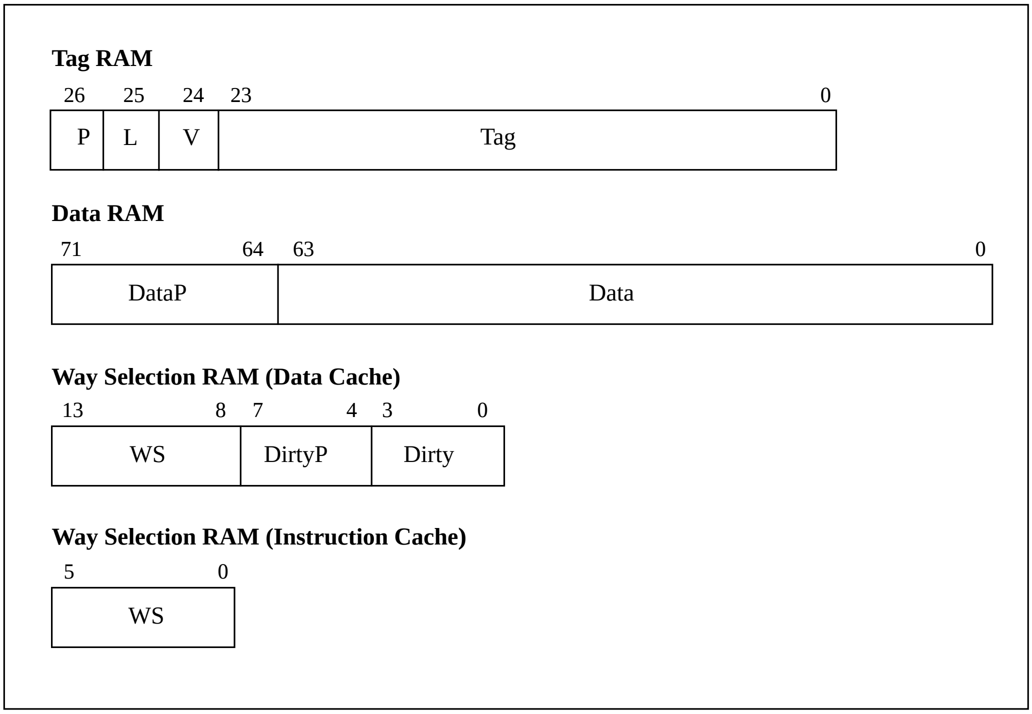


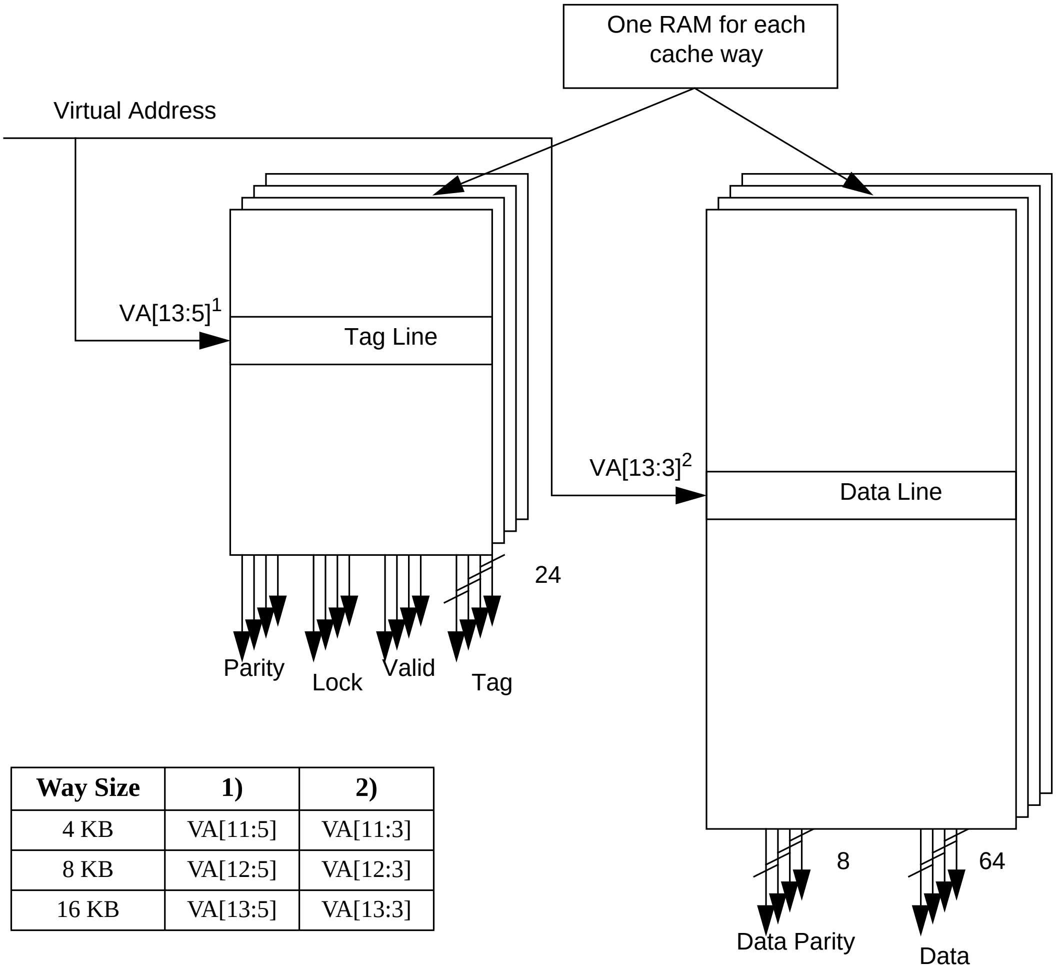
Figure 8-2 Cache RAM Formats

The Tag RAM contains the physical address bits that are used in the comparison for hit calculation and cache-line selection. It also contains control/status information for the cache line—the Valid bit is used to indicate if the line is valid, the Lock bit permits the line to be locked, and the optional Parity bit allows the detection of parity errors.

Each location in the Data RAM contains a doubleword (8 bytes) of cache data, along with optional even parity bits for each data byte. Note that each cache line, containing four doublewords (32 bytes), has optional even parity bits for each data byte.

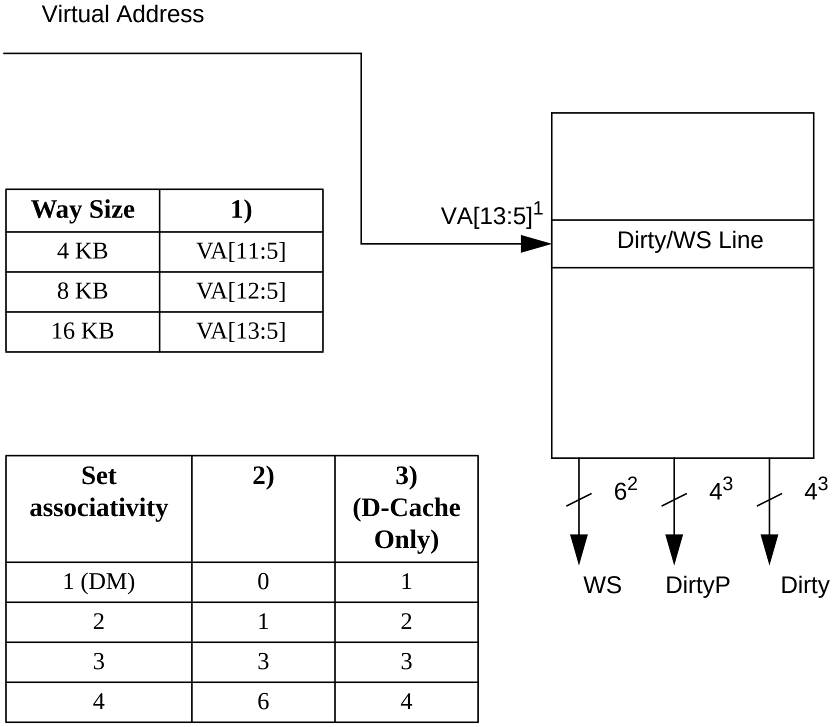
The WS RAM contains a status field managed by the way-selection logic that is used to implement the LRU-based algorithm for efficient cache-line replacements. The WS RAMs in the data cache contain a 6-bit status for the WS algorithm, 4 Dirty bits, and 4 associated optional even parity bits (one for each way). The WS RAMs in the instruction cache contain only a 6-bit status for the WS algorithm. Refer to Section 8.6, "Way Selection Algorithm" on page 149 and Figure 8-4 on page 146.

When the cache is accessed, bits in the virtual address are used to index both Tag and Data RAMs, as shown in Figure 8-3 and Figure 8-4. The number of virtual address bits that are used depends on the cache way size. When the physical address is available from the MMU, it is compared with the physical address tag(s) indexed by the virtual address, and, if there is a match, the cache data is delivered and the WS status is updated.



Way Size	1)	2)
4 KB	VA[11:5]	VA[11:3]
8 KB	VA[12:5]	VA[12:3]
16 KB	VA[13:5]	VA[13:3]

Figure 8-3 Cache Data and Tag Indexing



Way Size	1)
4 KB	VA[11:5]
8 KB	VA[12:5]
16 KB	VA[13:5]

Set associativity	2)	3) (D-Cache Only)
1 (DM)	0	1
2	1	2
3	3	3
4	6	4

Figure 8-4 Way Selection Indexing

When there is an I-cache or D-cache miss, the cache controller initiates a four doubleword refill to the way selected by the WS algorithm. The cache controllers are capable of processing hits during refills and uncached loads.

8.2.1 Instruction Cache Access

The virtual address for the instruction fetch is available prior to the instruction pipeline’s I Stage and is used to index the I-Data RAMs, I-Tag RAMs, and I-WS RAM, as explained in the previous section. A read of the I-Data RAM and I-Tag RAM is initiated at the beginning of the I Stage to determine if the required instruction resides in the cache. When the MMU indicates that the physical address is available, it is compared with the 1 to 4 different tags from the I-Tag RAMs. If there is an ITLB hit, the address will be available in the second half of the I Stage. The calculated hit information will be available at the end of the I Stage, and in case of a hit, the instruction will be available in the beginning of the D Stage. Information generated by the WS algorithm is written to the WS RAM after the instruction is returned.

Figure 8-5 shows an example of an instruction load that generates an ITLB hit as well as a cache hit. For simplicity, the figure shows only one of the up to 4 doublewords loaded from the cache data arrays.

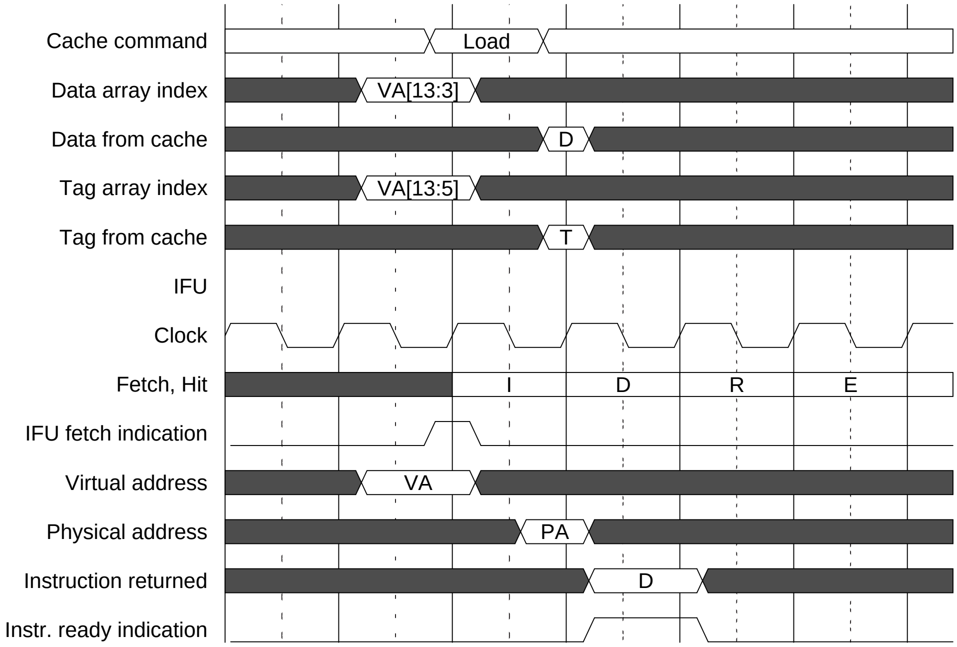


Figure 8-5 Example of Instruction Fetch with ITLB Hit and Cache Hit

8.2.2 Data Cache Access

The virtual address for a data load is available prior to the falling clock edge of the E Stage and is used to index the D-Data RAMs, D-Tag RAMs, and D-WS RAM. A read of the D-Data RAM and D-Tag RAM is initiated on the falling clock edge of the E Stage to determine if the required data resides in the cache. When the MMU indicates that the physical address is available, it is compared with the 1 to 4 different tags from the D-Tag RAMs. If there is a DTLB hit, the address will be available in the first half of the M Stage. The calculated hit information will be available prior to the falling edge in the M Stage, and for a hit, the data will be available in the last half of the M Stage. Information generated by the WS algorithm is written to the WS RAM after the data is returned.

Figure 8-6 shows an example of a data load that generates a DTLB hit as well as a cache hit. For simplicity, the figure shows only one of the up to 4 doublewords loaded from the data arrays.

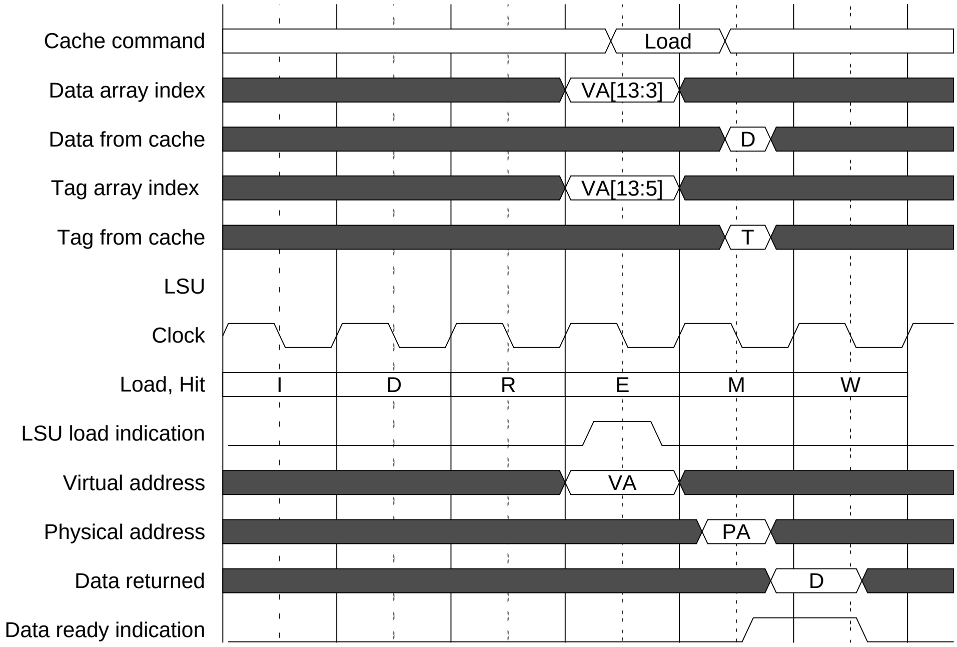


Figure 8-6 Example of Data Load with DTLB Hit and Cache Hit

8.3 Cache Write Policies

The MMU selects the cache write policy based on the virtual address (refer to Chapter 4, “Memory Management,” for further information). The 5K core supports the following four cache write policies:

- Cacheable, Write through, No Write allocate
- Cacheable, Write through, Write allocate
- Uncached (Write around)
- Cacheable, Write-back (Write allocate)

These policies are described in the following subsections.

8.3.1 Write Through, No Write Allocate

The cache is first searched to see if the target address is in the cache. If the target resides in the cache, the cache content is updated, and main memory is also written; the Dirty bit and Tag status bits are not modified. If the cache look-up misses, only main memory is written.

8.3.2 Write Through, Write Allocate

The cache is first searched to see if the target address is in the cache. If the target resides in the cache, the cache content is updated, and main memory is also written; the Dirty bit and Tag status bits are not modified. If the cache look-up misses, the line is refilled into the cache, the store data is merged with the refilled data, the Dirty bit is cleared (indicating the line is clean), and the store data is also sent to the write buffer. The cache line that is replaced by the refill (the so-called *victim*) is selected by the LRU algorithm (refer to Section 8.6, "Way Selection Algorithm" on page 149 and Figure 8-4 on page 146). If the line to be replaced is Dirty, it is evicted (written back to memory) before it is replaced.

8.3.3 Write Back, Write Allocate

The cache is first searched to see if the target address is in the cache. If the target resides in the cache, the cache content is updated, and the line is marked as Dirty. If the cache look-up misses, the line is refilled into the cache, the store data is merged with the refilled data, and the line is marked as Dirty. The least-recently used (LRU) cache line is replaced by the line refilled. If the line to be replaced is Dirty, it is evicted (written back to memory) before it is replaced.

8.3.4 Uncached

Addresses in memory areas designated as uncached are not read from the cache. Stores to these addresses are written directly to main memory, without modifying the cache contents.

If the memory is designated as uncached accelerated, the write buffer will, whenever possible, gather multiple stores and then initiate a burst write to memory. Uncached accelerated stores are described in Section 8.7, "Write Buffer".

8.4 Cached Loads and Fetches

For cached loads and instruction fetches, the cache is searched first, and if there is a cache miss, the data is read from main memory. The least-recently used cache line is evicted if it is Dirty and replaced by the new data.

8.5 Uncached Loads and Fetches

Uncached loads and instruction fetches always read data from main memory, and do not modify the cache contents.

8.6 Way Selection Algorithm

The 5K core uses a least-recently used (LRU) algorithm to select which cache line to replace (and possibly evict) on a cache miss. The algorithm is implemented by including a field for each index that encodes the order in which the ways have been accessed. These fields are stored in the WS RAM (refer to Figure 8-2).

The WS field of an index is updated as follows:

- When a cache hit is generated, the associated way is updated to be the most-recently used way in the WS field of the corresponding index. The order of the other ways relative to each other is unchanged.
- When a tag line is invalidated, the associated way is updated to be the least-recently used way in the WS field of the corresponding index. The order of the other ways relative to each other is unchanged.

On a cache miss, the WS field (and Dirty bits) of the corresponding index is read and decoded to select which way to refill (and evict if the associated Dirty bit is set).

PREF and PREFX instructions causing a line refill will set the state to most-recently used. Nudged lines will always be marked as least-recently used.

On a CACHE instruction, the WS field is updated as follows:

- **Index (Writeback) Invalidate:** Least-recently used.
- **Index Load Tag:** No update.
- **Index Store Tag, WST=0:** Most-recently used if valid bit is set in *TagLo* CP0 register. Least-recently used if valid bit is cleared in *TagLo* CP0 register.

- **Index Store Tag, WST=1:** Update the field with the contents of the *TagLo* CP0 register (refer to Section 8.10, "CACHE Instruction" for the valid values of this field).
- **Index Store Data:** No update.
- **Hit Invalidate:** Least-recently used if a hit is generated, otherwise unchanged.
- **Fill:** Most-recently used.
- **Hit (Writeback) Invalidate:** Least-recently used if a hit is generated, otherwise unchanged.
- **Hit Writeback:** No update.
- **Fetch and Lock:** Most-recently used.

Note that because the caches support hit-under-miss, the line that is chosen to be refilled (and possibly evicted) may not always be the least-recently used line at the precise moment the line is replaced. It is only guaranteed to be the least-recently used line at the moment the cache miss is detected.

No special action is needed to initialize the WS field, because that field will be valid after all the lines/ways in the cache have been invalidated. Initializing the Tag RAMs by using the CACHE Index Store Tag (WST=0) instruction to write zeros to the Tag RAMs will force the way to be marked as least-recently used.

8.7 Write Buffer

The write buffer is used to buffer all store transactions to the Bus Interface Unit (BIU). It contains a one-line (32-byte) eviction buffer used for cache-line write backs, as well as a four-doubleword entry store buffer used for write-through and uncached stores. For uncached accelerated stores, the store buffer has special features, as described below.

In order to avoid read-after-write hazards, the write buffer contains logic that checks the addresses of all write-buffer entries against load addresses from the data cache controller. If there is a conflict, the load is held until after the store has completed.

As soon as data is stored in the write buffer, a request for a store transaction is sent to the BIU.

The write buffer supports uncached accelerated stores by merging consecutive-word stores into a single doubleword store, and by gathering four doublewords for a single burst transaction. Note that the first doubleword of a burst must be aligned on a cache-line boundary (address bits [4:3] = 00). Note also that sequential address order is required in order to merge and gather uncached accelerated stores. The next address must be a) the same as the previous address (and the store have a valid merge pattern) to cause a merge, or b) the previous address incremented by 1 to continue the gathering process. A valid merge pattern requires that the store is an uncached accelerated store, and that two successive uncached accelerated stores have complementary byte enables—either the first store has byte enables 0F, and the following store is to the same address and has inverted byte enables (F0), or the first store has byte enables F0, and the following store is to the same address and has inverted byte enables (0F). When the merge/gather is complete, the write buffer will request a burst transaction to empty the buffer.

Merged/gathered data in the store buffer is transferred to the external interface buffer under one of the following conditions:

- Completion of merge/gather.
- A mergeable store is attempted from a non-sequential address.
- Execution of the SYNC instruction.
- Execution of the WAIT instruction.
- A store having an invalid merge pattern.
- Any store that is not an uncached accelerated store.

- Any load from outside accelerated address space.

8.8 Read Buffer

The read buffer is a four doubleword-deep FIFO, placed between the BIU and the data cache controller. It enables cache-line refills from the BIU to start immediately, even when the data cache controller has to complete a line eviction before it can receive data from the BIU—the read buffer can store the incoming data for the cache refill until the eviction has completed. However, if the data cache controller is ready to receive the refill data when it is available in the BIU, the FIFO is bypassed, and the data is forwarded directly from the BIU to the data cache controller.

8.9 Transaction Priority

When multiple transactions are requested, they are handled according to the priority (from highest to lowest priority) shown below.

1. Eviction buffer store on collision with a pending data load or refill.
2. Write buffer store on collision with a pending data load or refill.
3. Refill of a data cache line.
4. Uncacheable data load transaction.
5. Refill of an instruction cache line.
6. Uncacheable instruction load transaction.
7. Eviction buffer store.
8. Write buffer store.

Note that while a data load or refill waits for the write or eviction buffer store to finish its data phase on collision, an instruction load or refill or a new write can begin its address phase. The same is true when an uncached load waits for an uncached write to finish its data phase.

Uncached transactions are always handled in the same order as the order in which the instructions requesting the transactions are executed. This rule only applies to the true uncached transactions; that is, the uncached accelerated transactions are prioritized according to the list above.

8.10 CACHE Instruction

Both caches support the CACHE instructions, which allow users to manipulate the contents of the Data and Tag arrays, including the locking of individual cache lines. Note that before the CACHE instructions are allowed to execute, all initiated refills are completed and stores are sent to the write buffer. The CACHE instructions are described in detail in Chapter 12, “Instructions.” Caution: It is recommended not to lock all ways in cache as at least one way in the cache must be available for cache line refill to process a cache miss.

The CACHE Index Load Tag and Index Store Tag instructions can be used to read and write the WS RAM by setting the *WST* bit in the *ErrCtl* register. (The *ErrCtl* register is described in Section 6.23, "ErrCtl Register (CP0 Register 26, Select 0)" on page 134.) Note that when the *WST* bit is zero, the CACHE index instructions access the cache Tag array.

Not all values of the WS field are valid for defining the order in which the ways are selected. This is only an issue, however, if the WS RAM is written after the initialization (invalidation) of the Tag array. Valid WS field encodings for way selection order is shown in Table 8-1, Table 8-2, and Table 8-3.

Table 8-1 Way Selection Encoding, 4 Ways

Selection Order[a]	WS[5:0]	Selection Order	WS[5:0]
0123	000000	2013	100010
0132	000001	2031	110010
0213	000010	2103	100110
0231	010010	2130	101110
0312	010001	2301	111010
0321	010011	2310	111110
1023	000100	3012	011001
1032	000101	3021	011011
1203	100100	3102	011101
1230	101100	3120	111101
1302	001101	3201	111011
1320	101101	3210	111111

[a] The order is indicated by listing the least-recently used way to the left and the most-recently used way to the right, etc.

Table 8-2 Way Selection Encoding, 3 Ways

Selection Order[a]	WS[5:0][b]	Selection Order	WS[5:0]
012	0??00?	120	1??10?
021	0??01?	201	1??01?
102	0??10?	210	1??11?

[a] The order is indicated by listing the least-recently used way to the left and the most-recently used way to the right, etc.

[b] A '?' indicates a don't care when written and unpredictable when read.

Table 8-3 Way Selection Encoding, 2 Ways

Selection Order[a]	WS[5:0][b]	Selection Order	WS[5:0]
01	???0??	10	???1??

[a] The order is indicated by listing the least-recently used way to the left and the most-recently used way to the right, etc.

[b] A '?' indicates a don't care when written and unpredictable when read.

8.11 PREF and PREFX Instructions

The data cache controller supports the Prefetch instructions, PREF and PREFX, which are used to increase performance by informing the processor that the specified data is likely to be accessed or that it can be removed from the cache.

The load, load_streamed, and load_retained hint values are all handled as loads. The store, store_streamed, and store_retained hint values are all handled as stores with write allocation. Table 8-4 shows the action taken according to the hints.

Table 8-4 Action on PREF and PREFX Instructions

Cache Hit/Miss	PREF/PREFX Hint		
	load, load_streamed, load_retained	**store, store_streamed, store_retained**	**Nudge**
Hit	No action	No action	Evict if dirty and invalidate
Miss	Refill the line	Refill the line	No action

The PREF and PREFX instructions are described in detail in Chapter 12, “Instructions.”

8.12 Error Handling

The 5K core supports error handling for parity checks on cache lines errors that occur as a result of internal cache operations as well as errors that occur as a result of external memory operations.

8.12.1 Parity

The cache subsystem has optional support for parity checking. The description in this section applies only to systems with parity.

All parity bits are even. The cache Data RAMs contain byte parity bits, with 32 parity bits per line. The cache Tag has one parity bit per line for the 24-bit Tag, the Valid bit, and the Lock bit combined, and one parity bit for the Dirty bit. (Evictions of dirty lines are carried out regardless of the parity.) When the Data or Tag has a parity error, the error is signaled and the cache controller continues normal operation. For a Tag parity error, the *CacheErr* register’s *ET* bit is set, and for a Data parity error, the *CacheErr* register’s *ED* bit is set. For more information on parity error indication, refer to Section 6.23, "ErrCtl Register (CP0 Register 26, Select 0)" on page 134.

In general, the way indication when a cache error occurs is only valid for data RAM parity errors. The index indication is always valid, but the lower bits ([4:0]) are only valid for data RAM parity errors. Tag RAM parity errors are detected when there is cache hit or miss (instruction and data cache). Data RAM parity errors are detected when there is a cache hit (instruction and data cache) and during eviction (data cache only). Dirty bit parity errors are only detected when there is cache line replacement (data cache only).

Uncached transactions will never cause cache parity errors.

The invalidation of cache lines with parity errors is the responsibility of software.

All cache instructions except Index Load Tag (WST=0 or WST=1) and Index Store Tag (WST=0 or WST=1) can cause cache error exceptions when a parity error is detected.

To avoid using lines with hardware errors (for example, lines with bits that are ‘stuck’ at a particular value), the line must be locked and then invalidated. If the Valid bit is stuck at 1, the Tag must be set to an unused address. If the parity bit is stuck, the Tag must be adjusted accordingly (since parity must be set correctly to avoid future parity errors).

In some cases of cache error detection, it is impossible to determine (by examining cache line status bits or the *CacheErr* register) which process was corrupted by the error. In such cases, the fatal error indication (*EF* bit) is set in the *CacheErr* register. Refer to Section 6.24, "CacheErr Register (CP0 Register 27, Select 0)" for details. The following errors are indicated as fatal:

- Dirty parity error in dirty victim (Dirty bit cleared in Tag).
- Tag parity error in dirty victim.
- Data parity error in dirty victim.
- Write back store miss and WS field error at the requested index.
- Dual/Triple errors from different transactions, for example, a scheduled and a non-scheduled load.

8.12.2 WS Field Error

The bits used for the LRU algorithm are not parity protected, because they do not affect the correctness of the data loaded from the cache. However, when cache lines are refilled on a cache miss, the processor checks that one and only one way is selected when finding the least-recently used way. If this check fails and parity is enabled, a Cache Error exception is signaled and the *EW* bit in the *CacheErr* register is set. Refer to Section 6.24, "CacheErr Register (CP0 Register 27, Select 0)" on page 135 for details on error indication.

All values of the WS field not shown in Table 8-1, Table 8-2, and Table 8-3 cause a WS Field Error except those listed in Table 8-5. The values shown in Table 8-5 define a LRU way/line, though the order of the other three ways/lines are undefined. Thus a subsequent update of the field might cause the entire WS field to be valid, without the invalid part of the field having caused any errors or unnecessary exceptions.

Table 8-5 Invalid WS Fields Not Causing Errors

Selection Order[a][b]	WS[5:0]	Selection Order	WS[5:0]
0???	000011	2???	101010
0???	010000	2???	110110
1???	001100	3???	011111
1???	100101	3???	111001

[a] The order is indicated by listing the least-recently used way to the left and the most-recently used way to the right, etc.

[b] A '?' indicates undefined order.

When a way is locked, the bits in the WS field which contain information about the position of that particular way in the LRU queue are ignored and will not cause any errors for the line. Table 8-6 lists the bits in the WS field that are associated with each of the ways.

Table 8-6 Association of Ways and Bits in the WS Field

Way	Associated WS Bits
0	2, 3, 5
1	1, 2, 4
2	0, 1, 5
3	0, 3, 4

Note that a Cache Error exception is not caused by locking all the ways (so that no way is selected).

8.12.3 Bus Errors

When bus errors are detected, a Bus Error exception is generated for processing by CP0. Bus errors can be precise or imprecise, as explained in Section 5.11, "Bus Error Exception" on page 90. Data bus errors are imprecise and thus generate imprecise bus error exceptions (DBE) for processing by CP0. Bus errors on instruction fetch are precise.

In general, only bus errors on streamed instructions or data cause exceptions. If one or more bus errors is signaled during a cache refill operation, the cache line state is set to invalid and the line is unlocked. Bus errors on any doubleword during a write allocation will also cause Bus Error exceptions.

Bus errors that occur during any kind of store operation (single store, bursts, evictions) cause exceptions. This rule applies to the execution of the CACHE, PREF, and PREFX instructions as well as during normal operation.

Bus errors that occur during refills initiated by PREF and PREFX do not cause exceptions.

Chapter 9

Power Management

Chapter 8 describes the power management features of the 5K processor core, including active power management and power-down modes of operation. It contains the following sections:

- Section 9.1, "Register-Controlled Power Management"
- Section 9.2, "Instruction-Controlled Power Management"

9.1 Register-Controlled Power Management

The 5K processor core provides a standard software mechanism for placing the system into a low-power state, using the Reduced Power (*RP*) bit in the *CP0 Status* register. Setting the *RP* bit causes the 5K core to assert the *SI_RP* signal, which indicates to an external agent that the device is ready to be placed in power-down mode. The external agent can then decide whether to reduce the clock frequency and place the core into power-down mode. Other than its effect on the *SI_RP* signal, the *RP* bit has no effect internally in the 5K core.

Two additional bits in the *CP0 Status* register, *Exception Level* (*EXL*) and *Error Level* (*ERL*), support the power management function by informing an external agent of the occurrence of an exception or error while the core is in a low-power state.

The occurrence of an interrupt exception causes the *EXL* bit to be set, which in turn causes the assertion of the *SI_EXL* signal on the external bus, indicating to the external agent that an exception has occurred. If the processor is currently in reduced-power mode (*SI_RP* is HIGH), the external agent can choose to speed up the clocks in order for the exception to be serviced quickly.

Similarly, the occurrence of an error exception causes the *ERL* bit to be set, which in turn causes the assertion of the *SI_ERL* signal on the external bus, indicating to the external agent that an error has occurred. If the processor is currently in reduced-power mode (*SI_RP* is HIGH), the external agent can choose to speed up the clocks.

9.2 Instruction-Controlled Power Management

The second mechanism for invoking power-down mode is through execution of the WAIT instruction. If the bus is idle when the WAIT instruction reaches the M stage of the pipeline, the internal clocks are suspended and the pipeline is frozen. If the bus is not idle when the WAIT instruction reaches the M stage, the pipeline stalls until the bus becomes idle, at which time the clocks are stopped. However, the internal timer continues counting, and some of the input pins —*SI_Int*[5:0], *SI_NMI*, *SI_Reset*, *SI_ColdReset*, and *EJ_DINT*— continue to function normally.

Executing the WAIT instruction causes the assertion of the *SI_SLEEP* signal, which indicates to external agents that the device is in low-power mode. When 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. The CPU returns to the instruction-controlled power management mode when the next WAIT instruction is executed.

Chapter 10

EJTAG Debug Features

This chapter describes the EJTAG debug features supported by the 5K processor cores. It contains the following sections:

- Section 10.1, "Introduction"
- Section 10.2, "EJTAG Processor Extensions"
- Section 10.3, "Debug Control Register"
- Section 10.4, "Hardware Breakpoints"
- Section 10.5, "EJTAG Test Access Port"

Note that Section 10.5, "EJTAG Test Access Port" on page 188 does not describe the TAP signal-level interface.

10.1 Introduction

EJTAG is a hardware/software subsystem that provides comprehensive debugging and performance tuning capabilities to MIPS microprocessors and to system-on-a-chip components having MIPS processors. It exploits the infrastructure provided by the IEEE 1149.1 JTAG Test Access Port (TAP) standard to provide an external interface, and extends the MIPS instruction set and privileged resource architectures to provide a standard software architecture for integrated system debugging.

The following documents have background information for the description in this chapter:

- “EJTAG Specification”, rev. 02.60 or later, MIPS Technologies document number MD00047.
- “EJTAG Implementation Application Note”, rev. 1.00 or later, MIPS Technologies document number MD00071.
- IEEE Std. 1149.1-1990, “IEEE Standard Test Access Port and Boundary-Scan Architecture”

10.1.1 EJTAG Components and Options

EJTAG hardware support consists of several distinct components: extensions to the 5K processor, the EJTAG Test Access Port, the Debug Control Register, and the Hardware Breakpoint Unit. Figure 10-1 shows the relationship between these components in the EJTAG implementation. Some components and features are optional, and are implemented based on the needs of an implementation.

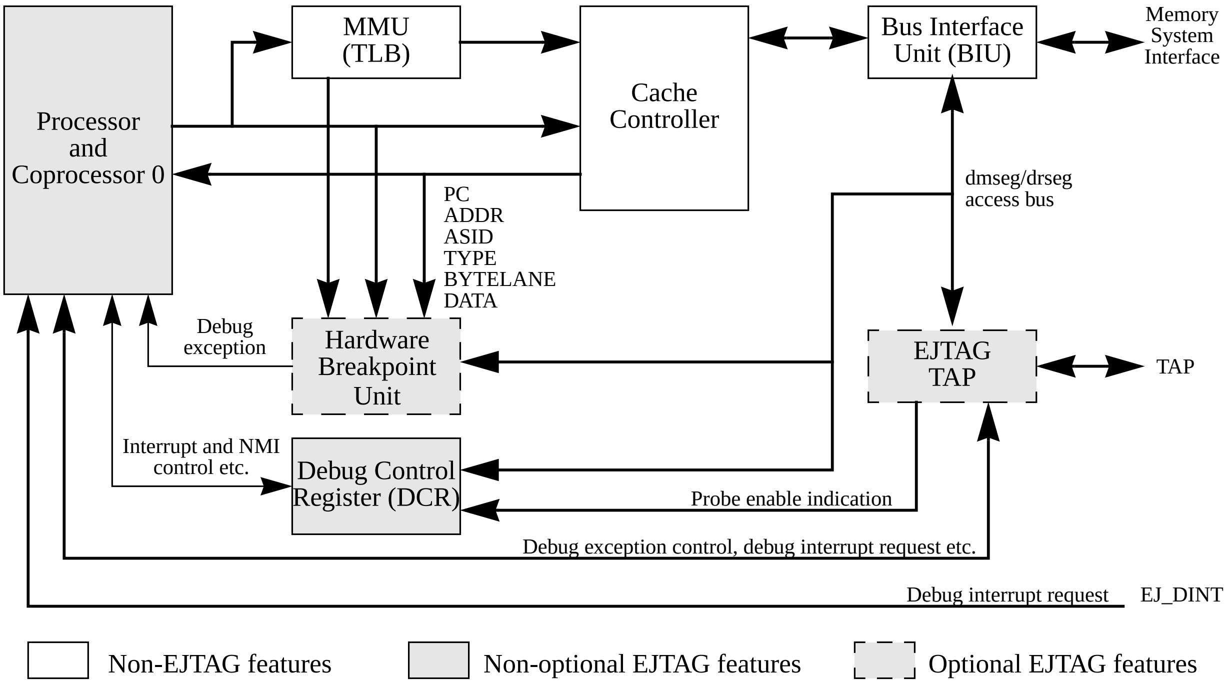


Figure 10-1 Simplified Block Diagram of EJTAG Components

Refer to the configurability description in the *MIPS64 5K Processor Core Family Implementor's Guide* to determine which optional blocks are included.

10.1.1.1 EJTAG Extensions to the MIPS Processor

The processor supports EJTAG-specific instructions, additional system coprocessor (CP0) registers, a single-step mode of execution, and vectoring to debug exceptions, which puts the processor in a special Debug Mode of execution, as described in Section 10.2, "EJTAG Processor Extensions" on page 162.

10.1.1.2 Debug Control Register

The Debug Control Register (DCR) is a memory-mapped register that is provided as part of the processor. It indicates the availability and status of EJTAG features. The memory-mapped region containing the DCR is available to software only in Debug Mode.

Refer to Section 10.3, "Debug Control Register" on page 169 for more information on the DCR.

10.1.1.3 Hardware Breakpoint Unit

The optional Hardware Breakpoint Unit implements memory-mapped registers that control the instruction and data hardware breakpoints. The memory-mapped region containing the hardware breakpoint registers is accessible to software only in Debug Mode.

If hardware breakpoints are a part of the implementation, then the following functionalities are provided:

- Four independent instruction hardware breakpoints
- Two independent data hardware breakpoints

The presence or absence of hardware breakpoint capability is indicated to debug software in the DCR.

Refer to Section 10.4, "Hardware Breakpoints" on page 171 for more information on the DCR.

10.1.1.4 EJTAG Test Access Port

The optional EJTAG Test Access Port (TAP) provides a standard JTAG (IEEE 1149.1) TAP interface to the EJTAG system. The TAP is necessary for all TAP-based EJTAG capabilities to allow host-based debugging and processor access to external debug memory. The presence or absence of off-board EJTAG memory is indicated to debug software via the DCR.

An implementation without a TAP implicitly disallows the EJTAG memory and TAP system access capabilities, but allows the remaining EJTAG services (Debug Mode, single-step, and software and hardware breakpoints) while executing from RAM or ROM.

Refer to Section 10.5, "EJTAG Test Access Port" on page 188 for more information on the TAP.

10.1.2 Register and Memory Map Overview

This subsection summarizes the registers and special memory that are used for the EJTAG debug solution. The locations of more details on these registers and memory locations are indicated below.

10.1.2.1 Coprocessor 0 Register

The Coprocessor 0 (CP0) registers consist of three registers related to debug features. These registers are accessible through the debug software executed on the processor; they provide debug control and status information. General information about the debug CP0 registers is found in Section 10.2.4, "EJTAG Coprocessor 0 Registers" on page 165.

10.1.2.2 Memory-Mapped EJTAG Register

The memory-mapped EJTAG registers are located in the debug register segment (drseg), which is a sub-segment of the debug segment (dseg). Debug software accesses these registers when the processor is executing in Debug Mode. These registers provide both miscellaneous debug control and control of hardware breakpoints. General information about the debug segment and registers is found in Section 10.2.5, "Debug Mode Address Space" on page 166 and Section 10.2.5.2, "Access to drseg (EJTAG Registers) Address Range" on page 167.

The following registers are present in the drseg:

- Debug Control Register (DCR), see Section 10.3, "Debug Control Register" on page 169
- Instruction hardware breakpoint registers (if hardware breakpoints are implemented), see Section 10.4.6, "Instruction Breakpoint Registers" on page 180
- Data hardware breakpoint registers (if hardware breakpoints are implemented), see Section 10.4.7, "Data Breakpoint Registers" on page 183

10.1.2.3 Memory-Mapped EJTAG Memory

The memory-mapped EJTAG memory is located in the debug memory segment (dmseg), which is a subsegment of the debug segment (dseg). Debug software accesses this segment when the processor is executing in Debug Mode. The EJTAG probe handles all accesses to this segment through the Test Access Port (TAP), whereby the processor has access to dedicated debug memory even if no debug memory was originally located in the system. The transactions made through the memory-mapped EJTAG memory are denoted processor accesses, and is shown as an example in Section 10.5.3, "Example of EJTAG Memory Access through Processor Access" on page 202.

General information about the debug segment and memory is found in Section 10.2.5, "Debug Mode Address Space" on page 166.

10.1.2.4 EJTAG Test Access Port Registers

The probe accesses EJTAG Test Access Port (TAP) registers (shown in Table 10-5 on page 166) through the TAP, so the processor cannot access these registers. These registers allow specific control of the target processor through the TAP. General information about the TAP registers is found in Section 10.5.2, "TAP Data Registers" on page 190.

10.1.3 Register Field Notations

Table 10-1 defines the R/W0 and R/W1 read/write notations used in the descriptions of the debug registers.

Table 10-1 Register Field Read/Write Notations

Read/Write Notation	Hardware Interpretation	Software Interpretation
R/W0	Similar to the R/W interpretation, except a software write of value 1 to this bit is ignored.	
R/W1	Similar to the R/W interpretation, except a software write of value 0 to this bit is ignored.	

10.2 EJTAG Processor Extensions

This section gives an overview of the processor’s EJTAG debug behavior. Some features are described elsewhere in this manual, in which case cross references to their descriptions are used; however, some information is duplicated in order to give a more complete view of the debug features. The 5K processor core extensions for EJTAG provide the following features which are always available:

- Debug exceptions, see Section 10.2.1, "Debug Exceptions" on page 162
- Debug Mode execution, see Section 10.2.2, "Debug Mode Execution" on page 163
- Debug Mode handling of processor resources, see Section 10.2.3, "Debug Mode Handling of Processor Resources" on page 163
- EJTAG CP0 registers: Debug, DEPC, and DESAVE, see Section 10.2.4, "EJTAG Coprocessor 0 Registers" on page 165
- Debug Mode address space with memory-mapped debug segment (dseg), see Section 10.2.5, "Debug Mode Address Space" on page 166
- Interrupt and NMI control from Debug Control Register (DCR), see Section 10.2.6, "Interrupts and NMIs" on page 168
- Reset issues, see Section 10.2.7, "Reset and Soft Reset of Processor" on page 168
- Debug interrupt request (EJ_DINT) signal, described in the “EJTAG Interface” chapter of *MIPS64 5K Processor Core Family Integrator’s Guide* document.

10.2.1 Debug Exceptions

Exceptions that causes the processor to go from Non-Debug Mode to Debug Mode are described in Section 6.1, "Index Register (CP0 Register 0, Select 0)" on page 105 and Section 5.23, "Debug Exceptions" on page 96. For example, an exception can occur on an SDBBP (Software Debug Breakpoint) instruction or due to a Debug Single Step exception, described in Section 5.23.6, "Debug Single Step Exception" on page 98.

10.2.2 Debug Mode Execution

Debug Mode is entered only through a debug exception. It is exited as a result of either execution of a DERET instruction or application of a Reset or Soft Reset exception.

When the processor is operating in Debug Mode it has access to the same resources, instructions, and CP0 registers as in Kernel Mode. Restrictions on Kernel Mode access (non-zero coprocessor references, access to extended addressing controlled by UX, SX, KX, etc.) apply equally to Debug Mode, but Debug Mode provides some additional capabilities as described in this chapter.

Kernel Mode, Supervisor Mode, and User Mode are collectively considered as Non-Debug Mode. Debug software can determine if the processor is in Non-Debug Mode or Debug Mode through the DM bit in the Debug register.

10.2.3 Debug Mode Handling of Processor Resources

Unless otherwise specified, the processor resources in Debug Mode are handled identically to those in Kernel Mode. Some identical cases are described in the following subsections for emphasis.

In addition, see the following related sections for more information:

- Section 10.2.6, "Interrupts and NMIs" for handling in both Debug and Non-Debug Modes
- Section 10.2.7, "Reset and Soft Reset of Processor" for handling in both Debug and Non-Debug Modes

10.2.3.1 Debug Mode Instruction Set

The full native ISA of the processor is accessible in Debug Mode.

Use the DERET (Debug Exception Return) instruction to return to Non-Debug Mode from Debug Mode.

Coprocessor loads and stores to the dseg segment are not supported. The operation of the processor is UNDEFINED if a coprocessor load or store to dseg is executed in Debug Mode.

10.2.3.2 Debug Mode Exceptions

Exceptions that can occur in Debug Mode are described in Section 5.23.8, "Handling of Exceptions in Debug Mode" on page 99.

10.2.3.3 Coprocessors

A Debug Mode Coprocessor Unusable exception is raised under the same conditions as for a Coprocessor Unusable exception in Kernel Mode (see Section 10.2.1, "Debug Exceptions" on page 162). Therefore Debug Mode software cannot reference Coprocessors 1 through 2 without first setting the respective enable in the Status register.

10.2.3.4 Random Register

The Random register is running in Debug Mode.

10.2.3.5 Count Register

The Count register is running in Debug Mode.

10.2.3.6 WatchLo/WatchHi Registers

The WatchLo/WatchHi registers (CP0 Registers 18 and 19) are inhibited from matching any instruction executed in Debug Mode.

10.2.3.7 Load Linked (LL/LLD) and Store Conditional (SC/SCD) Instruction Pair

A DERET instruction does not clear the LLbit, neither does the occurrence of a debug exception. Loads and stores to uncacheable locations that do not match the physical address of the previous LL instruction do not affect the result of SC instruction. The value of the LLbit is not directly visible by software.

10.2.3.8 SYNC Instruction Behavior Related to EJTAG Debug

Use the SYNC instruction to request the hardware to commit certain operations before proceeding. For example, a SYNC is required to remove memory hazards on reference to dseg. Similarly, a SYNC combined with the appropriate spacing (see Section 10.2.3.9, "CP0 and dseg Hazards" on page 164) is used to remove Coprocessor 0 (CP0) hazards. For example, the SYNC instruction ensures that status bits in the Debug register and the hardware breakpoint registers are fully updated before the debug handler accesses them and before Debug Mode is exited.

The SYNC instruction provides specific behavior as described in Table 10-2.

Table 10-2 SYNC Instruction References

Behavior	Section References
Commit accesses to dseg	See Section 10.2.5, "Debug Mode Address Space" on page 166
Update the DDBLImpr bits in the Debug register	See Section 5.23.1, "Exception Handling of Debug Exceptions" on page 96 and Section 6.20, "Debug Register (CP0 Register 23, Select 0)" on page 126
Update the BS bits in the IBS and DBS registers	See Section 10.4.4.3, "Imprecise Debug Exception Caused by Data Breakpoint" on page 178
Update the DBusEP, CacheEP, and MCheckP bits in the Debug register	See Section 6.20, "Debug Register (CP0 Register 23, Select 0)" on page 126

The SYNC instruction must be executed before leaving Debug Mode in order to commit all accesses to dseg, for example, to commit accesses to set up hardware breakpoints.

10.2.3.9 CP0 and dseg Hazards

Because resources controlled via Coprocessor 0 and EJTAG memory and registers in dseg affect the operation of various pipeline stages of the processor, manipulation of these resources might produce results that are not detectable by subsequent instructions for some number of execution cycles. When no hardware interlock exists between one instruction that causes an effect that is visible to a second instruction, a CP0 or dseg hazard exists.

Table 10-3 lists the spacing required to allow the consumer to eliminate the hazard. The values in the “Required” Spacing column represent spacing that the debug handler code must insert. The values are specific for the 5K core. The general 5K core hazard table is listed in Table 12-1 on page 218.

Table 10-3 “Required” CP0 and dseg Hazard Spacing

Producer	→	Consumer	Hazard On	“Required” Spacing (Cycles)
SYNC	→	DERET	dseg memory locations	2
SYNC	→	Load / Store	BSn bits in the IBS and DBS registers in drseg	2
SYNC	→	MFC0 Debug	$\text{Debug}_{\text{DDBLImpr}}$, $\text{Debug}_{\text{DBusEP}}$, $\text{Debug}_{\text{CacheEP}}$, $\text{Debug}_{\text{MCheckP}}$	2
MTC0 DEPC	→	DERET	DEPC	0
MTC0 Debug	→	DERET	Debug	0
MTC0 Debug[LSNM]	→	Load / Store in dseg	Debug[LSNM]	3
MTC0 Debug[IEXI]	→	Instructions that can cause an imprecise exception	Debug[IEXI]	3

Dependencies from the SYNC instruction as producer take effect because specific updates of dseg memory and resolving of pending imprecise exception indications are triggered by the SYNC instruction (refer to Section 10.2.3.8, "SYNC Instruction Behavior Related to EJTAG Debug" on page 164).

Use an SSNOP instruction should be used for each inserted spacing cycle, because the SSNOP instruction (executes like a NOP) is defined to convert instruction issues to cycles in a superscalar design, in which case the same debug code runs on a superscalar MIPS implementation.

10.2.4 EJTAG Coprocessor 0 Registers

The three Coprocessor 0 registers for EJTAG are shown in Table 10-4.

Table 10-4 Coprocessor 0 Registers for EJTAG

Register Number	Sel	Register Mnemonic	Function	Reference
23	0	Debug	Debug indications and controls for the processor, including information about recent debug exception.	See Section 6.20, "Debug Register (CP0 Register 23, Select 0)" on page 126
24	0	DEPC	Debug Exception Program Counter with address of last debug exception or exception in Debug Mode.	See Section 6.21, "Debug Exception Program Counter Register (CP0 Register 24, Select 0)" on page 129

Table 10-4 Coprocessor 0 Registers for EJTAG (Continued)

Register Number	Sel	Register Mnemonic	Function	Reference
31	0	DESAVE	Debug Exception Save scratchpad register available for the debug handler.	See Section 6.30, "Debug Exception SAVE (DESAVE) (CP0 register 31)" on page 140

10.2.5 Debug Mode Address Space

Debug Mode access to unmapped address space is identical to that of Kernel Mode. Mapped areas are accessible as in Kernel Mode, but only if a valid translation is possible immediately by the MMU. A memory access that would cause a TLB-type exception if tried from Kernel Mode causes re-entry into Debug Mode (see Section 10.2.3.2, "Debug Mode Exceptions" on page 163) through an exception if the memory access is tried while in Debug Mode. Memory accesses causing TLB-type exceptions are therefore not handled by the usual memory management routines if these memory accesses are made while in Debug Mode.

Updating and handling of cached areas is the same as that in Kernel Mode.

In addition, an uncached and unmapped debug segment dseg (EJTAG area) appears in the address range 0xFFFF FFFF FF20 0000 to 0xFFFF FFFF FF3F FFFF. The dseg appears in the kseg part of the compatibility segment, but access to kseg is still possible as described in the subsections below. Coprocessor loads and stores to dseg are not allowed. Table 10-5 shows the dseg subdivision and attributes.

Table 10-5 Physical Address and Cache Attribute for dseg, dmseg and drseg

Segment Name	Subsegment Name	Virtual Address	Reference Address	Cache Attribute
dseg	dmseg	0xFFFF FFFF FF20 0000 to 0xFFFF FFFF FF2F FFFF	Because the dseg address range is serviced exclusively by the EJTAG features, there are no physical address per se. Instead the lower 21 bits of the virtual address select the appropriate reference in either EJTAG memory or registers. References are not mapped through the TLB, nor do the accesses appear on the external system memory interface.	Uncached
	drseg	0xFFFF FFFF FF30 0000 to 0xFFFF FFFF FF3F FFFF		

The SYNC instruction, followed by appropriate spacing as described in Section 10.2.3.9, "CP0 and dseg Hazards" on page 164, must be executed to ensure that an access to dseg is committed (for example, after writing to dseg and before leaving Debug Mode). This procedure ensures that locations in dseg are fully updated for Non-Debug Mode, otherwise behavior of the processor is UNDEFINED.

10.2.5.1 Access to dmseg (EJTAG memory) Address Range

The probe services the dmseg segment, and the transactions made through the memory-mapped EJTAG memory in dmseg are denoted as processor accesses. Table 10-6 shows the behavior of processor accesses in Debug Mode to the dmseg address range from 0xFFFF FFFF FF20 0000 to 0xFFFF FFFF FF2F FFFF.

Table 10-6 Access to dmseg Address Range

Transaction	ProbEn bit in DCR register	LSNM bit in Debug Register	Access
Fetch	1	x	dmseg
	0	x	See comments below regarding behavior when ProbEn is 0
Load/Store	1	0	dmseg
		1	Kernel Mode address space
	0	1	Kernel Mode address space
		0	See comments below regarding behavior when ProbEn is 0
'x' denotes don't care.			

From Table 10-6, when ProbEn equals 0 for dmseg accesses, debug software accessed dmseg when the ProbEn bit was 0, indicating that there is no probe available to service the request. Debug software must read the state of the ProbEn bit in the DCR register before attempting to reference dmseg. However, accessing dmseg while ProbEn is 0 can occur because there is an inherent race between the debug software sampling the ProbEn bit as 1 and the probe clearing it to 0. The probe can therefore not assume that a reference to dmseg never occurs if the ProbEn bit is dynamically cleared to 0. If debug software references dmseg when ProbEn is 0, the reference hangs until it is satisfied by the probe.

The protocol for accesses to dmseg does not allow a transaction to be aborted once started, except by a Reset or Soft Reset exception. If the TAP is not present in the implementation, then the operation of the processor is UNDEFINED if the dmseg is accessed. Transactions of all sizes are allowed to dmseg.

10.2.5.2 Access to drseg (EJTAG Registers) Address Range

The drseg segment is used when the memory-mapped debug registers are accessed. Table 10-7 shows the behavior of processor accesses in Debug Mode to the drseg address range from 0xFFFF FFFF FF30 0000 to 0xFFFF FFFF FF3F FFFF.

Table 10-7 Access to drseg Address Range

Transaction	LSNM bit in Debug Register	Access
Fetch	x	Operation of the processor is UNDEFINED at fetch
Load/Store	0	drseg (see comments below the table)
	1	Kernel Mode address space
'x' denotes don't care.		

Instruction fetches from drseg are not allowed. The operation of the processor is UNDEFINED if the processor tries to fetch from drseg.

The DCR register, at offset 0x0000 in drseg, is always available but registers that set up hardware breakpoints are optional and only available if implemented. Debug software is expected to read the DCR register to determine if the hardware breakpoint 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 drseg.

The allowed transaction size is limited for drseg, as only doubleword size transactions are allowed. Operation of the processor is UNDEFINED for other transaction sizes.

10.2.6 Interrupts and NMIs

Interrupts and non-maskable interrupts (NMIs) are handled as described in the following subsections.

10.2.6.1 Interrupts

Interrupts are requested through either asserted external hardware signals or internal software-controllable bits. Interrupt exceptions are disabled when any of the following conditions are true:

- The processor is operating in Debug Mode
- The Interrupt Enable (IntE) bit in the Debug Control Register (DCR) is cleared (see Section 10.3, "Debug Control Register" on page 169)
- A non-EJTAG-related mechanism disables the interrupt exception

A pending interrupt is indicated through the Cause register, even if Interrupt exceptions are disabled.

10.2.6.2 NMIs

An NMI is requested on the asserting edge of the NMI signal to the processor, and an internal indicator holds the NMI request until the NMI exception is actually taken.

NMI exceptions are disabled when either of the following is true:

- The Processor is operating in Debug Mode
- The NMI Enable (NMIE) bit in the Debug Control Register (DCR) is cleared (see Section 10.3, "Debug Control Register" on page 169)

If an asserting edge on the NMI signal to the processor is detected while the NMI exception is disabled, then the NMI request is held pending and is deferred until NMI exceptions are no longer disabled.

A pending NMI is indicated in the NMIpend bit in the DCR even if NMI exceptions are disabled.

10.2.7 Reset and Soft Reset of Processor

This subsection covers the handling of issues with respect to Reset and Soft Reset exceptions. For EJTAG features, there is no difference between a Reset and a Soft Reset exception occurring to the processor; they behave identically in both Debug Mode and Non-Debug Mode.

10.2.7.1 EJTAGBOOT Feature

The EJTAGBOOT feature allows a Debug Interrupt exception to be generated immediately after a Reset or Soft Reset exception has occurred.

When EJTAGBOOT is indicated at the occurrence of a Reset or Soft Reset exception, a Debug Interrupt exception is taken and the debug handler is executed from the probe even if no instructions can be fetched from the reset handler. Control of EJTAGBOOT is described in Section 10.5.1.3, "EJTAGBOOT and NORMALBOOT Instructions" on page 189.

10.2.7.2 Processor Reset by Probe through Test Access Port

The PrRst bit in the EJTAG Control register is provided on the EJ_PrRst signal. The signal has no reset effect on the 5K core internally, but the external logic may apply reset throgh the ordinary reset signals for the core. If a reset occurs, then all parts of the system are reset; partial resets are not allowed.

10.2.7.3 Reset Occurred Indication through Test Access Port

The Rocc bit in the EJTAG Control register is set upon occurrence of a Reset or Soft Reset exceptions in order to indicate the event to the probe.

Refer to Section 10.5.2.5, "EJTAG Control Register (ECR) (TAP Instruction CONTROL or ALL)" on page 196 for more information on the EJTAG Control Register.

10.2.7.4 Soft Reset Enable

The optional Soft Reset Enable (SRstE) bit in the Debug Control Register (DCR) can mask the reset signal outside the processor used to generate a Soft Reset exception; the value of the bit is output on the EJ_SRstE signal.

10.2.7.5 Reset of Other Debug Features

The effect of Reset and Soft Reset exceptions also applies to reset of the following:

- Debug Control Register (DCR), see Section 10.3, "Debug Control Register" on page 169.
- Hardware Breakpoint, see Section 10.4, "Hardware Breakpoints" on page 171.
- Test Access Port (TAP) EJTAG Control Register, see Section 10.5, "EJTAG Test Access Port" on page 188.

10.3 Debug Control Register

The Debug Control Register (DCR) controls and provides information about debug issues. This register is always implemented. It is 64 bits wide and can only be accessed with doubleword load and stores. The DCR is located in the drseg at offset 0x0000.

The Debug Control Register (DCR) provides the following key features:

- Interrupt and NMI control when in Non-Debug Mode
- NMI pending indication
- Availability indicator of instruction and data hardware breakpoints

The DataBrk and InstBrk bits within the DCR indicate the types of hardware breakpoints implemented. Debug software is expected to read hardware breakpoint registers for additional information on the number of implemented breakpoints. Refer to Section 10.4, "Hardware Breakpoints" for descriptions of the hardware breakpoint registers.

Hardware and software interrupts can be disabled in Non-Debug Mode using the DCR’s IntE bit. This bit is a global interrupt enable that is used along with several other interrupt enables that enable specific mechanisms.

The NMI interrupt can be disabled in Non-Debug Mode using the DCR’s NMIE bit; a pending NMI is indicated through the NMIpend bit. Pending interrupts are indicated in the Cause register, and pending NMIs are indicated in the DCR register NMIpend bit, even when disabled. Hardware and software interrupts and NMIs are always disabled in Debug Mode.

The SRstE bit allows masking of the external signal that generates a Soft Reset exception; the value of the bit is output on the EJ_SRstE signal. A soft reset can be applied to the system based on different events, referred to as sources. It is implementation dependent which soft reset sources in a system can be masked by the SRstE bit. Soft reset masking can be applied to a soft reset source only if that source can be efficiently masked in the system. The result is no reset at all for any part of the system, if masked. If only a partial soft reset is possible, then that soft reset source is not to be masked, because a “half” soft reset might cause the system to fail or hang without warning. There is no automatic indication of whether the SRstE bit is effective.

The ProbEn bit reflects the state of the ProbEn bit from the EJTAG Control register (ECR). Through this bit, the probe can indicate to the debug software running on the CPU if it expects to service dmseg accesses.

Figure 10-2 shows the format of the DCR register; Table 10-8 describes the DCR register fields. The reset values in Table 10-8 take effect on both Reset and Soft Reset exceptions.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																															

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		ENM	0											Data Brk	Inst Brk	0											IntE	NMIE	NMI pend	SRstE	Prob En

Figure 10-2 DCR Register Format

Table 10-8 DCR Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
ENM	29	Endianess in which the processor is running in Kernel and Debug Modes: 0: Little endian 1: Big endian	R	Preset
DataBrk	17	Indicates if data hardware breakpoint is implemented: 0: No data hardware breakpoint implemented 1: Data hardware breakpoint implemented	R	Preset
InstBrk	16	Indicates if instruction hardware breakpoint is implemented: 0: No instruction hardware breakpoint implemented 1: Instruction hardware breakpoint implemented	R	Preset
IntE	4	Hardware and software interrupt enable for Non-Debug Mode, in conjunction with other disable mechanisms: 0: Interrupt disabled 1: Interrupt enabled depending on other enabling mechanisms	R/W	1
NMIE	3	Non-Maskable Interrupt (NMI) enable for Non-Debug Mode: 0: NMI disabled 1: NMI enabled	R/W	1
NMIpend	2	Indicates pending NMI: 0: No NMI pending 1: NMI pending	R	0

Table 10-8 DCR Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bits			
SRstE	1	Controls soft reset enable: 0: Soft reset masked for soft rest sources dependent on implementation 1: Soft reset is fully enabled Bit value is output on the processor signal EJ_SRstE.	R/W	1
ProbEn	0	Indicates value of the ProbEn value in the ECR register: 0: No access should occur to dmseg 1: Probe services accesses to dmseg Reads as zero if Test Access Port (TAP) is not implemented.	R	Same value as ProbEn in ECR
0	63:30, 28:18, 15:5	Must be written as zeros; return zeros on reads.	0	0

10.4 Hardware Breakpoints

The optional hardware breakpoints compare addresses and data of executed instructions, including data load/store accesses. Instruction breakpoints can be set even on addresses in ROM areas, and data breakpoints can cause debug exceptions on specific data accesses. Instruction and data hardware breakpoints are alike in many aspects, and are described in parallel in the following sections. When the term “breakpoint” is used in this section, then the reference is to a “hardware breakpoint”, unless otherwise explicitly noted.

10.4.1 Introduction

When hardware breakpoints are included in the implementation, then there are four instruction breakpoints and two data breakpoints. These breakpoints provide the following key features:

- Instruction breakpoints cause debug exceptions on executed instructions, both in ROM and RAM. Bit masking is provided for virtual address compares; masking of compares with ASID (optional) is also provided.
- Data breakpoints cause debug exceptions on data accesses. Bit masking is provided for virtual address compares, and masking of compares with ASID is provided. Data value compares allow masking at the byte level, and qualification on byte access and access type is possible.
- Registers for setup and control are memory mapped in drseg, accessible in Debug Mode only.

The following subsections provide details of instruction and data breakpoints.

10.4.1.1 Instruction Breakpoint Overview

Figure 10-3 shows a block diagram of the instruction breakpoint feature. This instruction breakpoint compares the virtual address (PC) and the ASID of the executed instruction with each instruction breakpoint, applying masks on address and ASID. When an enabled instruction breakpoint matches the PC and ASID, a debug exception and/or a trigger is generated, and an internal bit in an instruction breakpoint register is set to indicate that a match occurred.

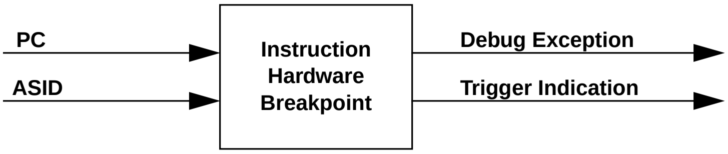


Figure 10-3 Instruction Breakpoint

10.4.1.2 Data Breakpoint Features

Figure 10-4 shows a block diagram of the data breakpoint feature. The data breakpoint compares the load or store access type (TYPE), the virtual address of the access (ADDR), the ASID, the accessed bytes (BYTELANE), and data value (DATA) with each data breakpoint, applying masks and/or qualifications on the access properties.

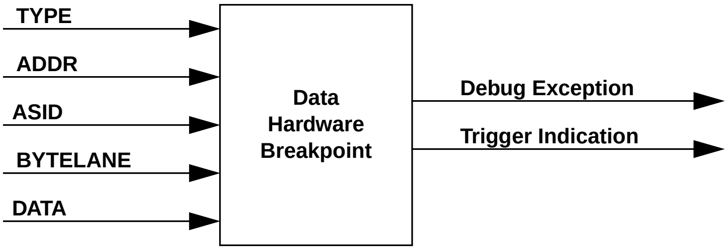


Figure 10-4 Data Breakpoint

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

10.4.2 Overview of Instruction and Data Breakpoint Registers

The InstBrk and DataBrk bits in the DCR register indicate whether breakpoints are implemented or not. If no breakpoints are implemented, then none of the registers associated with breakpoints are implemented; otherwise the registers shown in Table 10-9 and Table 10-10 are implemented.

Section 10.4.2.1, "Instruction Breakpoint Register Summary" and Section 10.4.2.2, "Data Breakpoint Register Summary" provide overviews of the instruction and data breakpoint registers, respectively. All registers are memory mapped in the drseg segment and are 64 bits wide.

10.4.2.1 Instruction Breakpoint Register Summary

Table 10-9 lists the Instruction Breakpoint registers. The Instruction Breakpoint Status register provides implementation indication and status for instruction breakpoints in general. The 4 implemented instruction breakpoints are numbered 0 to 3 for registers and breakpoints. The specific breakpoint number is indicated by “n”, and n is 0 to 3.

Table 10-9 Instruction Breakpoint Register Summary

Register Mnemonic	Register Name and Description	Reference
IBS	Instruction Breakpoint Status	See Section 10.4.6.1, "Instruction Breakpoint Status (IBS) Register" on page 180
IBAn	Instruction Breakpoint Address n	See Section 10.4.6.2, "Instruction Breakpoint Address n (IBAn) Register" on page 181

Table 10-9 Instruction Breakpoint Register Summary

Register Mnemonic	Register Name and Description	Reference
IBMn	Instruction Breakpoint Address Mask n	See Section 10.4.6.3, "Instruction Breakpoint Address Mask n (IBMn) Register" on page 181
IBASIDn	Instruction Breakpoint ASID n	See Section 10.4.6.4, "Instruction Breakpoint ASID n (IBASIDn) Register" on page 182
IBCn	Instruction Breakpoint Control n	See Section 10.4.6.5, "Instruction Breakpoint Control n (IBCn) Register" on page 182

Instruction Breakpoint register addresses are shown in Section 10.4.6, "Instruction Breakpoint Registers" on page 180.

10.4.2.2 Data Breakpoint Register Summary

Table 10-10 lists the Data Breakpoint Registers. The Data Breakpoint Status register provides implementation indication and status for data breakpoints in general. The two implemented data breakpoints are numbered 0 and 1 for registers and breakpoints. The specific breakpoint number is indicated by “n” and n is 0 or 1.

Table 10-10 Data Breakpoint Register Summary

Register Mnemonic	Register Name and Description	Reference
DBS	Data Breakpoint Status	See Section 10.4.7.1, "Data Breakpoint Status (DBS) Register" on page 184
DBAn	Data Breakpoint Address n	See Section 10.4.7.2, "Data Breakpoint Address n (DBAn) Register" on page 185
DBMn	Data Breakpoint Address Mask n	See Section 10.4.7.3, "Data Breakpoint Address Mask n (DBMn) Register" on page 185
DBASIDn	Data Breakpoint ASID n	See Section 10.4.7.4, "Data Breakpoint ASID n (DBASIDn) Register" on page 185
DBCn	Data Breakpoint Control n	See Section 10.4.7.5, "Data Breakpoint Control n (DBCn) Register" on page 186
DBVn	Data Breakpoint Value n	See Section 10.4.7.6, "Data Breakpoint Value n (DBVn) Register" on page 187

Data Breakpoint register addresses are shown in Section 10.4.7, "Data Breakpoint Registers" on page 183.

10.4.3 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 access. These conditions are described in the following subsections. A breakpoint only matches for instructions executed in Non-Debug Mode, never due to instructions executed in Debug Mode.

The match of an enabled breakpoint generates a debug exception as described in Section 10.4.4, "Debug Exceptions from Breakpoints" on page 177 and/or a trigger indication as described in Section 10.4.5, "Breakpoints Used as Triggerpoints" on page 179. The BE and/or TE bits in the IBCn or DBCn registers enable the breakpoints for breaks and triggers, respectively.

10.4.3.1 Conditions for Matching Instruction Breakpoints

When an instruction breakpoint is enabled, that breakpoint is evaluated in Non-Debug Mode with the instruction boundary address (the lowest address of a byte in the instruction) of every executed instruction. The instruction breakpoint is also evaluated on addresses usually causing an Address Error exception, a TLB exception, or other exceptions. It is thereby possible to cause a Debug Instruction Break exception on the destination address of a jump, even if a jump to that address would cause an Address Error exception. The breakpoint is not evaluated on instructions from speculative fetches or execution.

A match of an instruction breakpoint depends on a number of parameters, shown in Table 10-11. The fields in the instruction breakpoint registers are in the form $\text{REG}_{\text{FIELD}}$.

Table 10-11 Instruction Breakpoint Condition Parameters

Parameter	Description	Width
ASID	ASID field in EntryHi CP0 register.	8 bits
$\text{IBCn}_{\text{ASIDuse}}$	Use ASID value in compare for instruction breakpoint n: 0: Do not use ASID value in compare 1: Use ASID value in compare	1 bit
$\text{IBASIDn}_{\text{ASID}}$	Conditional Instruction breakpoint n ASID value for comparing.	8 bits
PC	Virtual address of instruction boundary or target for jump/branch.	64 bits
IBAn_{IBA}	Instruction breakpoint n address for compare with conditions.	64 bits
IBMn_{IBM}	Instruction breakpoint n address mask condition: 0: Corresponding address bit compared 1: Corresponding address bit masked	64 bits

The equation that determines the match is shown below with “C”-like operators. In the equation, 0 means all bits are 0’s, and ~0 means all bits are 1’s. The widths are similar to the widths of the parameters. The match equation is IB_match:

```
IB_match =
( ! IBCn_ASIDuse || ( ASID = = IBASIDn_ASID ) ) &&
( ( IBMn_IBM | ~ ( PC ^ IBAn_IBA ) ) = = ~0 )
```

The IB_match equation also applies when running in 32-bit addressing mode, in which case all 64 bits are compared between the PC and the IBAn_{IBA} register.

The match indication for instruction breakpoints is always precise; that is, it is indicated on the instruction causing the IB_match to be true.

10.4.3.2 Conditions for Matching Data Breakpoints

When a data breakpoint is enabled, that breakpoint is evaluated in Non-Debug Mode with both the access address of every data access due to load/store instructions (including loads/stores of coprocessor registers) and the address causing address errors upon data access. Data breakpoints are not evaluated with addresses from PREF (prefetch) or CACHE instructions.

A match of the data breakpoint depends on a number of parameters, shown in Table 10-12. The fields in the data breakpoint registers are in the form $\text{REG}_{\text{FIELD}}$.

Table 10-12 Data Breakpoint Condition Parameters

Reference	Description	Width
TYPE	Data access type is either load or store.	(no width)
$DBCn_{NoSB}$	Controls whether condition for data breakpoint is fulfilled on a store access: 0: Condition can be fulfilled on store access 1: Condition is never fulfilled on store access	1 bit
$DBCn_{NoLB}$	Controls whether condition for data breakpoint is fulfilled on a load access: 0: Condition can be fulfilled on load access 1: Condition is never fulfilled on load access	1 bit
ASID	ASID field in EntryHi CP0 register.	8 bits
$DBCn_{ASIDuse}$	ASID value used in compare for data breakpoint n: 0: Do not use ASID value in compare 1: Use ASID value in compare	1 bit
$DBASIDn_{ASID}$	Conditional Data breakpoint n ASID value for comparison.	8 bits
ADDR	Virtual address of data access.	64 bits
$DBAn_{DBA}$	Data breakpoint n address for compare with conditions.	64 bits
$DBMn_{DBM}$	Conditional Data breakpoint n address mask: 0: Corresponding address bit compared 1: Corresponding address bit masked	64 bits
BYTELANE	Byte lane access indication, where BYTELANE[0] is 1 only if the byte at bits [7:0] on the data bus is accessed, BYTELANE[1] is 1 only if the byte at bits [15:8] on the data bus is accessed, etc.	8 bits
$DBCn_{BAI}$	Determines whether access is ignored to specific bytes. BAI[0] ignores access to byte at bits [7:0] of the data bus, BAI[1] ignores access to byte at bits [15:8] of the data bus, etc.: 0: Condition depends on access to corresponding byte 1: Access for corresponding byte is ignored	8 bits
DATA	Data value from the data bus.	64 bits
$DBVn_{DBV}$	Conditional Data breakpoint n data value for comparison.	64 bits
$DBCn_{BLM}$	Conditional 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	8 bits

The match equations are shown below with “C”-like operators. In the equation, 0 means all bits are 0’s, and ~0 means all bits are 1’s. The bit widths are similar to the widths of the parameters.

DB_match is the overall match equation (the DB_addr_match, DB_no_value_compare, and DB_value_match equations in the DB_match equation are defined below):

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

DB_addr_match is defined as:

```
DB_addr_match =
( ! DBCn_ASIDuse || ( ASID = = DBASIDn_ASID ) ) &&
( ( DBMn$_{DBM}$ | ~ ( ADDR ^ DBAn$_{DBA}$ ) ) = = ~0 ) &&
( ( ~ DBCn$_{BAI}$ & BYTELANE ) != 0 )
```

The DB_addr_match equation also applies when running in 32-bit addressing mode, in which case all 64 bits are compared between the ADDR and the DBAn$_{DBA}$ field.

DB_no_value_compare is defined as:

```
DB_no_value_compare =
( ( DBCn$_{BLM}$ | DBCn$_{BAI}$ | ~ BYTELANE ) = = ~0 )
```

If a data value compare is indicated on a breakpoint, then DB_no_value_compare is false, and if there is no data value compare then DB_no_value_compare is true. Note that a data value compare is a run-time property of the breakpoint if (DBCn$_{BLM}$ | DBCn$_{BAI}$) is not ~0, because DB_no_value_compare then depends on BYTELANE provided by the load/store instructions.

If a data value compare is required, then the data value from the data bus is compared and masked with the registers for the data breakpoint, as shown in the DB_value_match equation:

```
DB_value_match =
( ( DATA[7:0] = = DBVn$_{DBV[7:0]}$ ) || ! BYTELANE[0] || DBCn$_{BLM[0]}$ || DBCn$_{BAI[0]}$ ) &&
( ( DATA[15:8] = = DBVn$_{DBV[15:8]}$ ) || ! BYTELANE[1] || DBCn$_{BLM[1]}$ || DBCn$_{BAI[1]}$ ) &&
......
( ( DATA[63:56] = = DBVn$_{DBV[63:56]}$ ) ||
    ! BYTELANE[7] || DBCn$_{BLM[7]}$ || DBCn$_{BAI[7]}$ )
```

Data breakpoints depend on endianess, because values on the byte lanes are used in the equations. Thus it is required that the debug software programs the breakpoints accordingly to endianess.

A precise match for a data breakpoint always occurs on data breakpoints on a store and on data breakpoints on a load if there is no data value compare. An imprecise match for a data breakpoint always occurs on a data breakpoint on a load with data value compare.

If a data value compare is required to evaluate a data breakpoint (the DB_no_value_compare equation is false), but a bus or cache error occurs on the load, then there is no valid data to use in the compare, and there will be no match in this case.

Unaligned addresses can result from explicit halfword, word, and doubleword accesses (for example, if an effective address of 0x01 is used as source of a Load Halfword (LH) instruction). The ADDR used in the comparison is the effective address. The BYTELANE value is defined according to Table 10-13.

Table 10-13 BYTELANE Value at Unaligned Address

Size	ADDR			BYTELANE[7:0]	
	[2]	[1]	[0]	Little Endian	Big Endian
Halfword	0	0	x	00000011_2	11000000_2
	0	1	x	00001100_2	00110000_2
	1	0	x	00110000_2	00001100_2
	1	1	x	11000000_2	00000011_2
Word	0	x	x	00001111_2	11110000_2
	1	x	x	11110000_2	00001111_2
Doubleword	x	x	x	11111111_2	
‘x’ denotes don’t care.					

With the above well-defined values of BYTELANE, the behavior is well-defined for data breakpoints without value compares on operations with unaligned addresses. The BLM field in the DBCn register can be used to avoid value compares if all BLM bits are set to 1.

If the data breakpoint depends on a value compare, then loads will cause an Address Error exception, and for stores the data value (DATA) is UNPREDICTABLE. This UNPREDICTABLE data can cause match of a data breakpoint on a store, but an implementation can be configured to never indicate a match on data breakpoints in this case.

If a debug exception is taken on the store then the debug handler can investigate the processor state and thereby determine if the address was unaligned and UNPREDICTABLE store data for the memory access thereby caused the debug exception. If a debug exception is not taken for the store, then an Address Error exception is taken.

If the data breakpoint is used as a triggerpoint (see Section 10.4.5, "Breakpoints Used as Triggerpoints" on page 179) then a BS bit might be set after a compare with UNPREDICTABLE data; however, an Address Error exception occurs in this case.

10.4.4 Debug Exceptions from Breakpoints

This subsection describes how to set up instruction and data breakpoints to generate debug exceptions when the match conditions are true.

10.4.4.1 Debug Exception Caused by Instruction Breakpoint

When the BE bit in the IBCn register is set, instruction breakpoints are enabled. A Debug Instruction Break exception occurs when the IB_match equation is true (see Section 10.4.3.1, "Conditions for Matching Instruction Breakpoints" on page 174). The corresponding BS bit in the IBS register is set when the breakpoint generates the debug exception.

The Debug Instruction Break exception is precise, so the DEPC register and DBD bit in the Debug register point to the instruction that caused the IB_match equation to be true. Refer to Section 6.21, "Debug Exception Program Counter Register (CP0 Register 24, Select 0)" on page 129.

The instruction receiving the debug exception only updates the debug-related registers. That instruction will not cause any loads/stores to occur. Thus a debug exception from a data breakpoint cannot occur at the same time an instruction receives 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 must disable the breakpoint when returning to the instruction, otherwise the Debug Instruction Break exception will reoccur. An alternative is for debug software to emulate the instruction(s) in software and change the DEPC accordingly.

10.4.4.2 Precise Debug Exception Caused by Data Breakpoint

The BE bit in the DBCn register must be set for data breakpoints to be enabled. A debug exception occurs when the DB_match condition is true (see Section 10.4.3.2, "Conditions for Matching Data Breakpoints" on page 174).

A Debug Data Break Load/Store exception occurs when a data breakpoint indicates a precise match. In this case, the DEPC register and DBD bit in the Debug register point to the load/store instruction that caused the DB_match equation (see Section 10.4.3.2, "Conditions for Matching Data Breakpoints" on page 174) to be true, and the corresponding BS bit in the DBS register is set. For the 5K processor, these precise debug exceptions only occur for data breakpoints without data value compare on load instructions and both with and without a data value compare on store instructions. Table 10-14 shows details about behavior of the instruction causing the debug exception.

Table 10-14 Behavior on Precise Exceptions from Data Breakpoints

Data Breakpoint and Instruction	Load/Store Instruction Execution	Destination Register	External Memory System Access
Store wo/w value match	Not completed	Not updated[a]	Store to memory is not committed
Load without value match		Not updated[b]	Load from memory does not occur

[a] Applies to the Store Conditional Word/Double (SC/SCD) instructions

[b] Includes side effects like for the Load Linked Word/Double (LL/LLD) instructions

The rules shown in Table 10-15 describe updates of the BS bits when several data breakpoints match the same access and generate a debug exception.

Table 10-15 Rules for Updating BS Bits on Precise Exceptions from Data Breakpoints

Instruction	Breakpoints That Match...		Update of BS Bits for Matching Data Breakpoints	
	Without Value Compare	With Value Compare	Without Value Compare	With Value Compare
Load / Store	One or more	None	BS bits set for all	(No matching breakpoints)
Load	One or more	One or more	BS bits set for all	Unchanged BS bits since load of data value does not occur, so match of the breakpoint cannot be determined
Load	None	One or more	(No matching breakpoints)	Covered by imprecise debug exception, see Section 10.4.4.3, "Imprecise Debug Exception Caused by Data Breakpoint" on page 178.
Store	One or more	One or more	BS bits set for all	BS bits set for all
Store	None	One or more	(No matching breakpoints)	BS bits set for all

Any BS bit set prior to the match and debug exception is kept set, since only debug software can clear the BS bits.

The debug handler usually returns to the instruction that caused the Debug Data Break Load/Store exception, whereby the instruction is re-executed. Debug software must disable breakpoints when returning to the instruction, otherwise the Debug Data Break Load/Store exception will reoccur. An alternative is for debug software to emulate the instruction in software and change the DEPC accordingly.

10.4.4.3 Imprecise Debug Exception Caused by Data Breakpoint

The BE bit in the DBCn register must be set for data breakpoints to be enabled. A debug exception occurs when the DB_match condition is true (see Section 10.4.3.2, "Conditions for Matching Data Breakpoints" on page 174).

A Debug Data Break Load Imprecise exception occurs when a data breakpoint indicates an imprecise match. In this case, the DEPC register and DBD bit in the Debug register point to an instruction later in the execution flow rather than at the load that caused the DB_match equation to be true. For the 5K processor, these imprecise debug exceptions only occur for data breakpoints with data value compares on load instructions.

The load instruction causing the Debug Data Break Load Imprecise exception always updates the destination register and finalizes the access to the external memory system. Therefore this load instruction is not re-executed on return from the debug handler, because the DEPC register and DBD bit do not point to that instruction.

Imprecise data breakpoints can be pending due to an outstanding scheduled load. The breakpoints are then evaluated when the access finalizes, and a Debug Data Break Load Imprecise exception is generated, if there is a match. If a debug exception had already occurred at the time of the match (for example, due to a precise debug exception), then the match from the scheduled load causes the corresponding BS and DDBLImpr bits to be set, but no debug exception is generated since the processor is already in Debug Mode.

The debug handler is required to execute the SYNC instruction, followed by two cycles spacing (for example, using two SSNOP instructions), before the BS and DDBLImpr bits are accessed for reading or writing. This delay ensures that these bits are fully updated.

All BS bits set prior to the match and debug exception are kept set, because only debug software can clear the BS bits.

10.4.5 Breakpoints Used as Triggerpoints

Software can set up both instruction and data breakpoints such that a matching breakpoint does not generate a debug exception, but sends an indication through the BS bit only. The TE bit in the IBCn or DBCn register controls whether an instruction or data breakpoint, respectively, is used as a triggerpoint. Triggerpoints are evaluated for matches under the same criteria as breakpoints.

The BS bit in the IBS or DBS register is set for a triggerpoint when the respective IB_match condition (see Section 10.4.3.1, "Conditions for Matching Instruction Breakpoints" on page 174) or DB_match condition (see Section 10.4.3.2, "Conditions for Matching Data Breakpoints" on page 174) is true.

For the BS bit to be set for an instruction triggerpoint, either the instruction must be fully executed or an exception must occur on the instruction.

The BS bit for a data triggerpoint can only be set if no exception with higher priority than the Debug Data Break Load/Store exception with address match only occurred on the load/store instruction. For exceptions with equal or lower priority than the Debug Data Break Load/Store exception with address match only, the BS bits are still set for a matching triggerpoint. For example, the BS bit is set even if a TLB or Bus Error exception occurred on the load/store instruction. Data triggerpoints with value compares require the data value to be valid for the BS bit to be set, which is not the case if, for example, a TLB or Bus Error exception occurs on a load instruction. However, for stores, the trigger can compare on UNPREDICTABLE data as described in Section 10.4.3.2, "Conditions for Matching Data Breakpoints" on page 174.

Table 10-16 shows the rules for updating the BS bits.

Table 10-16 Rules for Updating BS Bits on Data Triggerpoints

Instruction	Without/With Value Compare	BS Bits Update for Triggerpoint
Load / Store	Without data value compare	BS bit is set if no exception with higher priority than the Debug Data Break Load/Store exception with address match only occurred on the instruction.
Load	With data value compare	BS bit is set if no exception with higher priority than the Debug Data Break Load exception with address + data value match occurred on the instruction.
Store	With data value compare	BS bit is set if no exception with higher priority than the Debug Data Break Load/Store exception with address match only occurred on the instruction. Note that setting the BS bit is UNPREDICTABLE for a data triggerpoint with a data value compare, in case an unaligned address results from the store data access, see Section 10.4.3.2, "Conditions for Matching Data Breakpoints" on page 174.

Data breakpoints with imprecise matches generate imprecise triggers when enabled by the TE bit.

10.4.6 Instruction Breakpoint Registers

This subsection describes the instruction breakpoint registers. These registers provide status and control for the instruction breakpoints. All registers are in drseg. The four implemented breakpoints are numbered 0 to 3 for registers and breakpoints. The specific breakpoint number is indicated by "n", with n in 0 to 3. Table 10-17 shows the registers and their respective address offsets.

Table 10-17 Instruction Breakpoint Register Mapping

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

To remove hazards when updating instruction breakpoint registers, the debug handler must execute the SYNC instruction followed by two cycles spacing (for example, using two SSNOPs) after writing to the instruction breakpoint registers and before leaving Debug Mode. This procedure ensures that the registers are fully updated for Non-Debug Mode, otherwise behavior of the processor is UNDEFINED.

10.4.6.1 Instruction Breakpoint Status (IBS) Register

The Instruction Breakpoint Status (IBS) register holds implementation and status information about the instruction breakpoints. It is located at drseg offset 0x1000. The ASIDsup bit applies to all instruction breakpoints. Figure 10-5 shows the format of the IBS register; Table 10-18 describes the IBS register fields.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																															

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

Figure 10-5 IBS Register Format

Table 10-18 IBS Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
ASIDsup	30	Indicates if ASID compare is supported in instruction breakpoints: 0: No ASID compare 1: ASID compare (IBASIDn register implemented) ASID support indication does not guarantee a TLB-type MMU, because the same breakpoint implementation can be used with processors having all different types of MMUs.	R	1
BCN	27:24	Number of instruction breakpoints implemented: 0: Reserved 1-15: Number of instructions breakpoints	R	4

Table 10-18 IBS Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bits			
BS[3:0]	3:0	Break Status (BS) bit for breakpoint n is at BS[n], where n is 0 to 3. A bit is set to 1 when the condition for its corresponding breakpoint has matched. The number of BS bits corresponds to the number of breakpoints indicated by the BCN field. Debug software is expected to clear the bits before use, because reset does not clear these bits.	R/W0	Undefined
0	63:31, 29:28, 23:4	Must be written as zeros; return zeros on read.	0	0

10.4.6.2 Instruction Breakpoint Address n (IBAn) Register

The Instruction Breakpoint Address n (IBAn) register has the address used in the condition for instruction breakpoint n. It is located at drseg offset 0x1100 + 0x100 * n. Figure 10-6 shows the format of the IBAn register; Table 10-19 describes the IBAn register field.

<table>
<tr><th>63</th><th>62</th><th>61</th><th>60</th><th>59</th><th>58</th><th>57</th><th>56</th><th>55</th><th>54</th><th>53</th><th>52</th><th>51</th><th>50</th><th>49</th><th>48</th><th>47</th><th>46</th><th>45</th><th>44</th><th>43</th><th>42</th><th>41</th><th>40</th><th>39</th><th>38</th><th>37</th><th>36</th><th>35</th><th>34</th><th>33</th><th>32</th></tr>
<tr><td colspan="32">IBAn</td></tr>
</table>

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="32">IBAn</td></tr>
</table>

Figure 10-6 IBAn Register Format

Table 10-19 IBAn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
IBA	63:0	Instruction breakpoint address for condition.	R/W	Undefined

10.4.6.3 Instruction Breakpoint Address Mask n (IBMn) Register

The Instruction Breakpoint Address Mask n (IBMn) register has the address compare mask used in the condition for instruction breakpoint n. The address that is masked is in the IBAn register. The IBMn register is located at drseg offset 0x1108 + 0x100 * n. Figure 10-7 shows the format of the IBMn register; Table 10-20 describes the IBMn register field.

<table>
<tr><th>63</th><th>62</th><th>61</th><th>60</th><th>59</th><th>58</th><th>57</th><th>56</th><th>55</th><th>54</th><th>53</th><th>52</th><th>51</th><th>50</th><th>49</th><th>48</th><th>47</th><th>46</th><th>45</th><th>44</th><th>43</th><th>42</th><th>41</th><th>40</th><th>39</th><th>38</th><th>37</th><th>36</th><th>35</th><th>34</th><th>33</th><th>32</th></tr>
<tr><td colspan="32">IBMn</td></tr>
</table>

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="32">IBMn</td></tr>
</table>

Figure 10-7 IBMn Register Format

Table 10-20 IBMn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
IBM	63:0	Instruction breakpoint address mask for condition: 0: Corresponding address bit compared 1: Corresponding address bit masked	R/W	Undefined

10.4.6.4 Instruction Breakpoint ASID n (IBASIDn) Register

The Instruction Breakpoint ASID n (IBASIDn) register has the ASID value used in the compare for instruction breakpoint n. It is located at drseg offset 0x1110 + 0x100 * n.

Figure 10-8 shows the format of the IBASIDn register; Table 10-21 describes the IBASIDn register fields.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																															

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

Figure 10-8 IBASIDn Register Format

Table 10-21 IBASIDn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
ASID	7:0	Instruction breakpoint ASID value for compare.	R/W	Undefined
0	63:8	Must be written as zeros; return zeros on read.	0	0

10.4.6.5 Instruction Breakpoint Control n (IBCn) Register

The Instruction Breakpoint Control n (IBCn) register determines what constitutes instruction breakpoint n: triggerpoint, breakpoint, ASID value inclusion. This register is located at drseg offset 0x1118 + 0x100 * n. Figure 10-9 shows the format of the IBCn register; Table 10-21 describes the IBCn register fields.

<table>
<tr><th>63</th><th>62</th><th>61</th><th>60</th><th>59</th><th>58</th><th>57</th><th>56</th><th>55</th><th>54</th><th>53</th><th>52</th><th>51</th><th>50</th><th>49</th><th>48</th><th>47</th><th>46</th><th>45</th><th>44</th><th>43</th><th>42</th><th>41</th><th>40</th><th>39</th><th>38</th><th>37</th><th>36</th><th>35</th><th>34</th><th>33</th><th>32</th></tr>
<tr><td colspan="32">0</td></tr>
</table>

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="8">0</td><td>ASIDuse</td><td colspan="20">0</td><td>TE</td><td>0</td><td>BE</td></tr>
</table>

Figure 10-9 IBCn Register Format

Table 10-22 IBCn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
ASIDuse	23	Use ASID value in compare for instruction breakpoint n: 0: Do not use ASID value in compare 1: Use ASID value in compare Debug software must only set ASIDuse if a TLB in the implementation is used by the application software.	R/W	Undefined
TE	2	Use instruction breakpoint n as triggerpoint: 0: Do not use it as triggerpoint 1: Use it as triggerpoint	R/W	0
BE	0	Use instruction breakpoint n as breakpoint: 0: Do not use it as breakpoint 1: Use it as breakpoint	R/W	0
0	63:24, 22:3, 1	Must be written as zeros; return zeros on read.	0	0

10.4.7 Data Breakpoint Registers

This subsection describes the data breakpoint registers. These registers provide status and control for the data breakpoints. All registers are in drseg. The two implemented breakpoints are numbered 0 and 1 for registers and breakpoints. The specific breakpoint number is indicated by “n”, with n as 0 or 1. The registers and their respective addresses offsets are shown in Table 10-23.

Table 10-23 Data Breakpoint Register Mapping

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

To remove hazards when updating data breakpoint registers, the debug handler must execute the SYNC instruction followed by at least two cycles in Debug Mode after writing to the data breakpoint registers (for example, using two SSNOPs). This procedure ensures that the registers are fully updated for Non-Debug Mode; otherwise behavior of the processor is UNDEFINED.

10.4.7.1 Data Breakpoint Status (DBS) Register

The Data Breakpoint Status (DBS) register holds implementation and status information about the data breakpoints. It is located at drseg offset 0x2000. The ASIDsup, NoSVmatch, and NoLVmatch fields apply to all data breakpoints. Figure 10-10 shows the format of the DBS register; Table 10-24 describes the DBS register fields.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																															

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

Figure 10-10 DBS Register Format

Table 10-24 DBS Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
ASIDsup	30	Indicates if ASID compare is supported in data breakpoints: 0: No ASID compare 1: ASID compare (DBASIDn register implemented) ASID support indication does not guarantee a TLB-type MMU, because the same breakpoint implementation can be used with processors having all different types of MMUs.	R	1
NoSVmatch	29	Indicates if a value compare on a store is supported in data breakpoints: 0: Data value and address in condition on store 1: Address compare only in condition on store	R	0
NoLVmatch	28	Indicates if a value compare on a load is supported in data breakpoints: 0: Data value and address in condition on load 1: Address compare only in condition on load	R	0
BCN	27:24	Number of data breakpoints implemented: 0: Reserved 1-15: Number of data breakpoints	R	2
BS[1:0]	1:0	Break Status (BS) bit for breakpoint n is at BS[n], where n is 0 or 1. The bit is set to 1 when the condition for its corresponding breakpoint has matched. The number of BS bits implemented corresponds to the number of breakpoints indicated by the BCN bit. Debug software must clear these bits before use, because they are not cleared by reset.	R/W0	Undefined
0	63:31, 23:2	Must be written as zeros; return zeros on read.	0	0

10.4.7.2 Data Breakpoint Address n (DBAn) Register

The Data Breakpoint Address n (DBAn) register has the address used in the condition for data breakpoint n. This register is located at drseg offset 0x2100 + 0x100 * n. Figure 10-11 shows the format of the DBAn register; Table 10-25 describes the DBAn register field.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
DBAn																															

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

Figure 10-11 DBAn Register Format

Table 10-25 DBAn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
DBA	63:0	Data breakpoint address for condition	R/W	Undefined

10.4.7.3 Data Breakpoint Address Mask n (DBMn) Register

The Data Breakpoint Address Mask n (IBMn) register has the address compare mask used in the condition for data breakpoint n. The address that is masked is in the DBAn register. The DBMn register is located at drseg offset 0x2108 + 0x100 * n. Figure 10-12 shows the format of the DBMn register; Table 10-26 describes the DBMn register field.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
DBMn																															

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

Figure 10-12 DBMn Register Format

Table 10-26 DBMn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
DBMn	63:0	Data breakpoint address mask for condition: 0: Corresponding address bit compared 1: Corresponding address bit masked	R/W	Undefined

10.4.7.4 Data Breakpoint ASID n (DBASIDn) Register

The Data Breakpoint ASID n (DBASIDn) register has the ASID value used in the compare for data breakpoint n. It is located at drseg offset 0x2110 + 0x100 * n.

Figure 10-13 shows the format of the DBASIDn register; Table 10-27 describes the DBASIDn register fields.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																															

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

Figure 10-13 DBASIDn Register Format

Table 10-27 DBASIDn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
ASID	7:0	Data breakpoint ASID value for compare.	R/W	Undefined
0	63:8	Must be written as zeros; return zeros on read.	0	0

10.4.7.5 Data Breakpoint Control n (DBCn) Register

The Data Breakpoint Control n (DBCn) register determines what constitutes data breakpoint n: triggerpoint, breakpoint, ASID value inclusion, load/store access fulfillment, ignore byte access, byte lane mask. This register is located at drseg offset 0x2118 + 0x100 * n. Figure 10-14 shows the format of the DBCn register; Table 10-28 describes the DBCn register fields.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																															

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								ASIDuse	0	BAI[7:0]								NoSB	NoLB	BLM[7:0]								0	TE	0	BE

Figure 10-14 DBCn Register Format

Table 10-28 DBCn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
ASIDuse	23	Use ASID value in compare for data breakpoint n: 0: Do not use ASID value in compare 1: Use ASID value in compare Debug software must only set ASIDuse if a TLB in the implementation is used by the application software.	R/W	Undefined
BAI[7:0]	21:14	Byte access ignore. Controls ignore of access to specific bytes. 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 Debug software must adjust for endianess when programming this field.	R/W	Undefined

Table 10-28 DBCn Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bits			
NoSB	13	Controls whether condition for data breakpoint is ever fulfilled on a store access: 0: Condition can be fulfilled on store access 1: Condition is never fulfilled on store access	R/W	Undefined
NoLB	12	Controls whether condition for data breakpoint is ever fulfilled on a load access: 0: Condition can be fulfilled on load access 1: Condition is never fulfilled on load access	R/W	Undefined
BLM[7:0]	11:4	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 Debug software must adjust for endianess when programming this field.	R/W	Undefined
TE	2	Use data breakpoint n as triggerpoint: 0: Do not use it as triggerpoint 1: Use it as triggerpoint	R/W	0
BE	0	Use data breakpoint n as breakpoint: 0: Do not use it as breakpoint 1: Use it as breakpoint	R/W	0
0	63:24, 22, 3, 1	Must be written as zeros; return zeros on read.	0	0

10.4.7.6 Data Breakpoint Value n (DBVn) Register

The Data Breakpoint Value n (DBVn) register has the value used in the condition for data breakpoint n. It is located at drseg offset 0x2120 + 0x100 * n. Figure 10-15 shows the format of the DBVn register; Table 10-29 describes the DBVn register field.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
DBVn																															

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

Figure 10-15 DBVn Register Format

Table 10-29 DBVn Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
DBV	63:0	Data breakpoint data value for condition. Debug software must adjust for endianess when programming this field.	R/W	Undefined

10.5 EJTAG Test Access Port

This section describes the EJTAG features provided when the optional EJTAG Test Access Port (TAP) is included in the implementation. The features are described in terms of the TAP Instruction register and data registers, where access to these registers through the 5K core TAP interface is described in the “EJTAG Interface” chapter of the *MIPS64 5K Processor Core Family Integrator’s Guide*.

The overall features of the EJTAG Test Access Port (TAP) are:

- Identification of device and EJTAG debug features accessed through the TAP
- EJTAG memory is in dseg, which provides a memory-mapped area handled by the probe through processor accesses, whereby the processor can execute a debug handler not present in the system memory
- Reset handling allows debug exception immediately after reset
- Debug interrupt request from probe
- Low-power mode indications
- Implementation-dependent processor and peripheral reset

If the TAP is not implemented, then other features depending on register values and indications from the TAP behaves as if these register values and indications have the power-up and reset values.

Note that all references to reset apply to processor reset or soft reset, unless otherwise explicitly stated.

10.5.1 Instruction Register and Special Instructions

The Instruction register controls selection of accessed data register(s), and controls the setting and clearing of the EJTAGBOOT indication.

The Instruction register is five bits wide. Table 10-30 shows the allocation of the TAP instruction.

Table 10-30 TAP Instruction Overview

Code	Instruction	Function
0x01	IDCODE	Selects Device Identification (ID) register
0x03	IMPCODE	Selects Implementation register
0x08	ADDRESS	Selects Address register
0x09	DATA	Selects Data register
0x0A	CONTROL	Selects EJTAG Control register
0x0B	ALL	Selects the Address, Data, and EJTAG Control registers

Table 10-30 TAP Instruction Overview (Continued)

Code	Instruction	Function
0x0C	EJTAGBOOT	Makes the processor take a debug exception after reset
0x0D	NORMALBOOT	Makes the processor execute the reset handler after reset
0x0E	FASTDATA	Selects the Data and Fastdata registers
0x1F	BYPASS	Selects Bypass register

The instructions IDCODE, IMPCODE, ADDRESS, DATA, CONTROL, and BYPASS select a single data register, as indicated in the table. The unused instructions select the Bypass register. The ALL, EJTAGBOOT, and NORMALBOOT instructions are described in the following subsections.

Any EJTAGBOOT indication must be cleared at power-up through a reset of the TAP. The “EJTAG Interface” chapter of the *MIPS64 5K Processor Core Family Integrator’s Guide* describes how this TAP reset must be performed using the EJ_TRST_N signal. At the TAP reset, the Instruction register is loaded with the IDCODE instruction.

10.5.1.1 ALL Instruction

The Address, Data and EJTAG Control data registers are selected at once with the ALL instruction, as shown in Figure 10-16, with connection between the TDI and TDO signals.

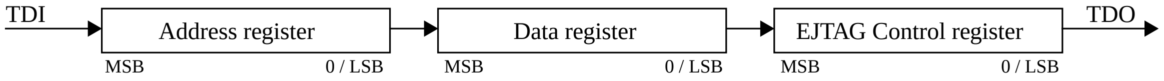


Figure 10-16 Selected Registers when ALL Instruction is Selected

10.5.1.2 FASTDATA Instruction

The Data and Fastdata registers are selected with the FASTDATA instruction, as shown in Figure 10-17, with connection between the TDI and TDO signals.

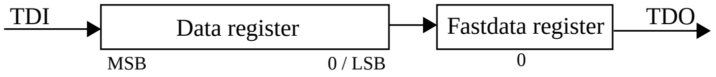


Figure 10-17 Selected Registers when FASTDATA Instruction is Selected

10.5.1.3 EJTAGBOOT and NORMALBOOT Instructions

The EJTAGBOOT and NORMALBOOT instructions control whether the processor takes a Debug Interrupt exception after reset with execution of the debug handler from the probe, or if it executes the reset handler as usual. An internal EJTAGBOOT indication holds information on the action to take at a processor reset, which applies to both reset and soft reset.

The internal EJTAGBOOT indication is set when the EJTAGBOOT instruction takes effect in the Update-IR state. The indication is cleared when the NORMALBOOT instruction takes effect in the Update-IR state, or when the Test-Logic-Reset state is entered (for example, when EJ_TRST_N signal is asserted low).

The internal EJTAGBOOT indication is cleared at power-up, so the processor executes the reset handler after power-up unless the EJTAGBOOT instruction is given through the TAP.

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

The EjtagBrk, ProbEn, and ProbTrap bits in the EJTAG Control register follow the internal EJTAGBOOT indication. They are all set at processor reset if a Debug Interrupt exception is to be generated, with execution of the debug handler from the probe.

When an EJTAGBOOT instruction is indicated at reset, then it is possible to take the debug exception and execute the debug handler from the probe even if no instructions can be fetched from the reset handler. This condition guarantees that the system will not hang in this type of case.

10.5.2 TAP Data Registers

Table 10-31 summarizes the data registers in the TAP. Complete descriptions of these registers are located in the following subsections.

Table 10-31 EJTAG TAP Data Registers

Instruction Used to Access Register	Register Name	Function	Reference
IDCODE	Device ID	Identifies device and accessed processor in the device.	See Section 10.5.2.1, "Device Identification (ID) Register (TAP Instruction IDCODE)" on page 191
IMPCODE	Implementation	Identifies main debug features implemented and accessible through the TAP.	See Section 10.5.2.2, "Implementation Register (TAP Instruction IMPCODE)" on page 192
DATA, FASTDATA or ALL	Data	Data register for processor access.	See Section 10.5.2.3, "Data Register (TAP Instruction DATA, FASTDATA or ALL)" on page 193
ADDRESS or ALL	Address	Address register for processor access.	See Section 10.5.2.4, "Address Register (TAP Instruction ADDRESS or ALL)" on page 195
CONTROL or ALL	EJTAG Control	Control register for most EJTAG features used through the TAP.	See Section 10.5.2.5, "EJTAG Control Register (ECR) (TAP Instruction CONTROL or ALL)" on page 196
FASTDATA	Fastdata	Fastdata register for fast processor access handling.	See Section 10.5.2.6, "Fastdata Register (TAP Instruction FASTDATA)" on page 200

Table 10-31 EJTAG TAP Data Registers (Continued)

Instruction Used to Access Register	Register Name	Function	Reference
BYPASS, EJTAGBOOT, NORMALBOOT, or unused EJTAG instructions	Bypass	Provides a one-bit shift path through the TAP.	See Section 10.5.2.7, "Bypass Register (TAP Instruction BYPASS, EJTAGBOOT, NORMALBOOT, or Unused)" on page 201

A read of a data register corresponds only to the Capture-DR state of the TAP controller, and a write of the data register corresponds to the Update-DR state only.

The initial states of these registers are specified with either a reset state or a power-up state. If a reset state is specified, then the indicated value is applied to the register when a processor reset is applied. If a power-up state is specified, then the indicated value is applied at power-up reset.

EJ_TCK does not have to be running in order for a processor reset to reset the registers.

10.5.2.1 Device Identification (ID) Register (TAP Instruction IDCODE)

The Device ID register is a 32-bit read-only register that identifies the specific device implementing EJTAG. This register is also defined in IEEE 1149.1. The Device ID register holds a unique number among different devices with EJTAG-compliant processors implemented. It is recommended that the register is also unique amongst different EJTAG-compliant processors in the same device.

Figure 10-18 shows the format of the Device ID register; Table 10-32 describes the Device ID register fields.

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

Figure 10-18 Device ID Register Format

Table 10-32 Device ID Register Field Descriptions

Fields		Description	Read/ Write	Power-up State
Name	Bits			
Version	31:28	Identifies the version of a specific device. The value in this field must be unique for particular values of Manufacturer ID and Part Number values. The value identifies a specific revision of the design (such as a sequence of bug fixes within the same major design). The value is assigned by the design house.	R	Preset
Part Number	27:12	Identifies the part number of a specific device. The value in this field must be unique for a particular Manufacturer ID value. Design houses wishing to use the MIPS Technologies, Inc. Manufacturer ID can request assignment of a group of Part Numbers. Once the numbers are assigned, the design house then manages those numbers. Assignment of Part Numbers within another Manufacturer ID value is done by the owner of that Manufacturer ID.	R	Preset

Table 10-32 Device ID Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Power-up State
Name	Bits			
ManufID	11:1	Identifies the manufacturer identity code of a specific device, which identifies the design house implementing the processor. According to IEEE 1149.1-1990 section 11.2, the manufacturer identity code is a compressed form of a JEDEC standard manufacturer's identification code in the JEDEC Publications 106, which can be found at: http://www.jedec.org/ ManufID[6:0] are derived from the last byte of the JEDEC code with the parity bit discarded. ManufID[10:7] provide a binary count of the number of bytes in the JEDEC code that contains the continuation character (0x7F). When the number of continuation characters exceeds 15, these four bits contain the modulo-16 count of the number of continuation characters. If the design house does not have a JEDEC Standard Manufacture's Identification Code, which is encoded for use in this field, the design house can request use of the MIPS Technologies, Inc. assigned number, or use the number assigned to the core provider. Use of the MIPS Technologies, Inc. number requires prior approval of the Director, MIPS Architecture. The MIPS Technologies, Inc. Standard Manufacturer's Identification Code is 0x127.	R	Preset
1	0	Ignored on write; returns one on read.	R	1

10.5.2.2 Implementation Register (TAP Instruction IMPCODE)

The Implementation register is a 32-bit read-only register that identifies features implemented in this EJTAG compliant processor, mainly those accessible from the TAP. Figure 10-19 shows the format of the Implementation register; Table 10-33 describes the Implementation register fields.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EJTAGver			R4k/R3k	0			DINTsup	0	ASIDsize		0				MIPS16	0	NoDMA	0													MIPS32/64

Figure 10-19 Implementation Register Format

Table 10-33 Implementation Register Field Descriptions

Fields		Description	Read/ Write	Power-up State
Name	Bits			
EJTAGver	31:29	Indicates the EJTAG version implemented. Se encoding description in the EJTAGver field in the CP0 Debug register Section 6.20, "Debug Register (CP0 Register 23, Select 0)" on page 126.	R	Same as for EJTAGver in CP0 Debug register.
DINTsup	24	Indicates support for the DINT signal from the probe: 0: DINT signal from the probe is not supported by this chip 1: Probe can use DINT signal to make debug interrupt on this chip	R	Determined by the EJ_DINTsup signal

Table 10-33 Implementation Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Power-up State
Name	Bits			
ASIDsize	22:21	Indicates size of the ASID field: 0: No ASID in implementation 1: 6-bit ASID 2: 8-bit ASID 3: Reserved	R	2
MIPS16	16	Indicates MIPS16™ ASE support in the processor: 0: No MIPS16 ASE support 1: MIPS16 ASE is supported	R	0
NoDMA	14	Indicates no EJTAG DMA support: 0: Reserved 1: No EJTAG DMA support	R	1
MIPS32/64	0	Indicates 32-bit or 64-bit processor: 0: 32-bit processor 1: 64-bit processor	R	1
0	28:25, 23, 20:17, 15, 13:1	Ignored on writes; return zeros on reads.	R	0

10.5.2.3 Data Register (TAP Instruction DATA, FASTDATA or ALL)

The read/write Data register is used for opcode and data transfers during processor accesses. The width of the Data register is 64 bits.

The value read in the Data register is valid only if a processor access for a write is pending, in which case the data register holds the store value. The value written to the Data register is only used if a processor access for a pending read is finished afterwards, in which case the data value written is the value for the fetch or load. This behavior implies that the Data register is not a memory location where a previously written value can be read afterwards.

Figure 10-20 shows the format of the Data register; Table 10-34 describes the Data register field.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Data																															

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

Figure 10-20 Data Register Format

Table 10-34 Data Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
Data	63:0	Data used by processor access.	R/W	Undefined

The contents of the Data register are not aligned but hold data as it is seen on a data bus for an external memory system. Thus the bytes are positioned in the Data register based on access size, address, and endianess.

The bytes not accessed for a processor access write are undefined, and the bytes not accessed for a processor access read can be written with any value by the probe shifting the value into the Data register.

Table 10-35 shows the byte positioning using the three LSBs of the Address register together with the Psz field from the EJTAG Control register. Byte 0 refers to bits 7:0, byte 1 refers to bits 15:8, and so on up to byte 7, which refers to bits 63:56, independent of endianess. The endianess is indicated through the ENM bit in the Debug Control Register (DCR), see Section 10.3, "Debug Control Register" on page 169.

Table 10-35 Data Register Contents

Psz from ECR	Size	Address[2:0]	Little Endian								Big Endian							
			7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	Byte	000_2								▨	▨							
		001_2							▨			▨						
		010_2						▨					▨					
		011_2					▨							▨				
		100_2				▨									▨			
		101_2			▨											▨		
		110_2		▨													▨	
		111_2	▨															▨
1	Halfword	000_2							▨	▨	▨	▨						
		010_2					▨	▨					▨	▨				
		100_2			▨	▨									▨	▨		
		110_2	▨	▨													▨	▨
2	Word	000_2					▨	▨	▨	▨	▨	▨	▨	▨				
	5-byte/Quinti	001_2				▨	▨	▨	▨	▨	▨	▨	▨	▨	▨			
	6-byte/Sexti	010_2			▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨		
	7-byte/Septi	011_2		▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	
	Word	100_2	▨	▨	▨	▨									▨	▨	▨	▨
	5-byte/Quinti	101_2	▨	▨	▨	▨	▨							▨	▨	▨	▨	▨
	6-byte/Sexti	110_2	▨	▨	▨	▨	▨	▨					▨	▨	▨	▨	▨	▨
	7-byte/Septi	111_2	▨	▨	▨	▨	▨	▨	▨			▨	▨	▨	▨	▨	▨	▨
3	Triple	000_2						▨	▨	▨	▨	▨	▨					
		010_2					▨	▨	▨			▨	▨	▨				
		100_2		▨	▨	▨									▨	▨	▨	
		110_2	▨	▨	▨											▨	▨	▨
	Doubleword	111_2	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
Reserved			n.a.								n.a.							

10.5.2.4 Address Register (TAP Instruction ADDRESS or ALL)

The read-only Address register provides the address for a processor access. The width of the register corresponds to the size of the physical address in the processor implementation, which is 36 bits.

The value read in the register is valid if a processor access is pending, otherwise the value is undefined.

The three LSBs of the register are used with the Psz field from the EJTAG Control register to indicate the size and data position of the pending processor access transfer. These bits are not taken directly from the address referenced by the load/store. See Section 10.5.2.3, "Data Register (TAP Instruction DATA, FASTDATA or ALL)" on page 193 for more details. Figure 10-21 shows the format of the Address register; Table 10-36 describes the Address register field.

35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Address																																			

Figure 10-21 Address Register Format

Table 10-36 Address Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
Address	35:0	Address used by processor access.	R	Undefined

10.5.2.5 EJTAG Control Register (ECR) (TAP Instruction CONTROL or ALL)

The 32-bit EJTAG Control Register (ECR) handles processor reset and soft reset indication, Debug Mode indication, access start, finish, and size and read/write indication. The ECR also:

- controls debug vector location and indication of serviced processor accesses,
- allows a debug interrupt request,
- indicates processor low-power mode, and
- allows implementation-dependent processor and peripheral resets.

The EJTAG Control register is not updated/written in the Update-DR state unless the Reset occurred (Rocc) bit is either already 0 or is written to 0 at the same time. This condition ensures proper handling of processor accesses after a reset.

Reset of the processor is indicated through the Rocc bit in the EJ_TCK clock signal domain a number of EJ_TCK clock cycles after it is removed in the processor clock domain in order to allow for proper synchronization between the two clock domains.

Figure 10-22 shows the format of the EJTAG Control register; Table 10-37 describes the EJTAG Control register fields.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rocc	Psz		0						Doze	Halt	PerRst	PRnW	PrAcc	0	PrRst	ProbEn	ProbTrap	0	EjtagBrk	0								DM	0		

Figure 10-22 EJTAG Control Register Format

Table 10-37 EJTAG Control Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
Rocc	31	Indicates if a processor reset or soft reset has occurred since the bit was cleared: 0: No reset occurred 1: Reset occurred The Rocc bit stays set as long as reset is applied. This bit must be cleared to acknowledge that the reset was detected. The EJTAG Control register is not updated in the Update-DR state unless Rocc is 0 or written to 0 at the same time, which ensures correct handling of the processor access after reset.	R/W0	1
Psz	30:29	Indicates the size of a pending processor access, in combination with the Address register: 0: Byte 1: Halfword 2: Word, 5-7 bytes 3: Triple, Doubleword A full description of this field is located in Section 10.5.2.3, "Data Register (TAP Instruction DATA, FASTDATA or ALL)" on page 193, including reserved combinations with Address register bits. This field is valid only when a processor access is pending, otherwise the read value is undefined.	R	Undefined
Doze	22	Indicates if the processor is in low-power mode: 0: Processor is not in low-power mode 1: Processor is in low-power mode Doze indicates Reduced Power (RP) and WAIT low-power modes.	R	0
Halt	21	Indicates if the internal system bus clock is running: 0: Internal system bus clock is running 1: Internal system bus clock is stopped Halt indicates a WAIT in the system that stops the internal system bus clock.	R	0
PerRst	20	Controls the peripheral reset with implementation-dependent behavior: 0: No peripheral reset applied 1: Peripheral reset applied The signal has no reset effect on the 5K core internally, but the external logic may apply reset throgh the ordinary reset signals for the core. There is no inherent indication of whether PerRst is effective, so the user must consult system documentation. When this bit is changed, then it is only guaranteed that the new value has taken effect when it can be read back here. This handshake mechanism ensures that the setting from the EJ_TCK clock domain takes effect in the processor clock domain and in peripherals. The value of the bit is output on the EJ_PerRst signal.	R/W	0

Table 10-37 EJTAG Control Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bits			
PRnW	19	Indicates read or write of a pending processor access: 0: Read processor access, for a fetch/load access 1: Write processor access, for a store access This value is defined only when a processor access is pending.	R	Undefined
PrAcc	18	Indicates a pending processor access and controls finishing of a pending processor access. When read: 0: No pending processor access 1: Pending processor access A write of 0 finishes a processor access if pending; otherwise operation of the processor is UNDEFINED if the bit is written to 0 when no processor access is pending. A write of 1 is ignored.	R/W0	0
PrRst	16	Controls the processor reset with implementation-dependent behavior: 0: No processor reset applied 1: Processor reset applied The signal has no reset effect on the 5K core internally, but the external logic may apply reset throgh the ordinary reset signals for the core. There is no inherent indication of an effective PrRst, so the user must consult system documentation. If a reset occurs on PrRst, then all parts of the system are reset. It is not allowed for only some device to be reset. When this bit is changed then it is guaranteed that the new value has taken effect when it can be read back here. This handshake mechanism ensures that the setting from the EJ_TCK clock domain takes effect in the processor clock domain and in peripherals. However, because a processor reset clears this bit, then the effect of setting it can be that the bit is cleared when the reset takes effect. In this case, the Rocc bit should be observed to detect that the reset took effect. The value of the bit is output on the EJ_PrRst signal.	R/W	0
ProbEn	15	Controls indication to the processor of whether the probe expects to handle accesses to EJTAG memory through servicing of processors accesses: 0: Probe does not service processors accesses 1: Probe will service processor accesses The ProbEn bit is reflected as a read-only bit in Debug Control Register (DCR) bit 0. When this bit is changed, then it is guaranteed that the new value has taken effect in the DCR when it can be read back here. This handshake mechanism ensures that the setting from the EJ_TCK clock domain takes effect in the processor clock domain. However, a change of ProbEn prior to setting the EjtagBrk bit is effective for the debug handler. Not all combinations of ProbEn and ProbTrap are allowed; see the description below this table.	R/W	See description below this table.

Table 10-37 EJTAG Control Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bits			
ProbTrap	14	Controls location of the debug exception vector: 0: Normal memory 0xFFFF FFFF BFC0 0480 1: EJTAG memory 0xFFFF FFFF FF20 0200 in dmseg When this bit is changed, then it is guaranteed that the new value is indicated to the processor when it can be read back here. This handshake mechanism ensures that the setting from the EJ_TCK clock domain takes effect in the processor clock domain. However, a change of ProbTrap prior to setting the EjtagBrk bit is effective at the debug exception. Not all combinations of ProbEn and ProbTrap are allowed; see the description below this table.	R/W	See description below this table.
EjtagBrk	12	Requests a Debug Interrupt exception to the processor when this bit is written as 1. The debug exception request is ignored if the processor is already in debug at the time of the request. A write of 0 is ignored. The debug request restarts the processor clock if the processor was in a low-power mode. The read value indicates a pending Debug Interrupt exception requested through this bit: 0: No pending Debug Interrupt exception requested through this bit 1: Pending Debug Interrupt exception Hardware clears this bit when the processor enters Debug Mode.	R/W1	See description below this table.
DM	3	Indicates if the processor is in Debug Mode: 0: Processor is in Non-Debug Mode 1: Processor is in Debug Mode	R	0
0	28:23, 17, 13, 11:4, 2:0	Must be written as zeros; return zeros on reads.	0	0

The reset value of the EjtagBrk, ProbTrap, and ProbEn bits follows the setting of the internal EJTAGBOOT indication. If the EJTAGBOOT instruction has been given, and the internal EJTAGBOOT indication is active, then the reset value of the three bits is set (1), otherwise the reset value is clear (0).

The results of setting these bits are:

- A Debug Interrupt exception is requested right after reset because EjtagBrk is set
- The debug handler is executed from the EJTAG memory because ProbTrap is set to indicate debug vector in EJTAG memory at 0xFFFF FFFF FF20 0200
- Service of the processor access is indicated because ProbEn is set

Thus it is possible to execute the debug handler right after reset, without executing any instructions from the normal reset handler.

Use of ProbTrap and ProbEn allows independent specification of the debug exception vector location and availability of EJTAG memory. Behavior for the different combinations is shown in Table 10-38. Note that not all combinations are allowed.

Table 10-38 Combinations of ProbTrap and ProbEn

ProbTrap	ProbEn	Debug Exception Vector	Processor Accesses
0	0	Normal memory at 0xFFFF FFFF BFC0 0480	Not serviced by probe
0	1		Serviced by probe
1	0	If these two bits are changed to this state, the operation of the processor is UNDEFINED, indicating that the debug exception vector is in EJTAG memory, but the probe will not service processor accesses.	
1	1	EJTAG memory at 0xFFFF FFFF FF20 0200	Serviced by probe

10.5.2.6 Fastdata Register (TAP Instruction FASTDATA)

The width of the Fastdata register is 1 bit. During a Fastdata access, the Fastdata register is written and read, i.e., a bit is shifted in and a bit is shifted out. During a Fastdata access, the Fastdata register value shifted in specifies whether the Fastdata access should be completed or not. The value shifted out is a flag that indicates whether the Fastdata access was successful or not (if completion was requested).

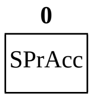


Figure 10-23 Fastdata Register Format

Table 10-39 Fastdata Register Field Description

Fields		Description	Read/ Write	Reset State
Name	Bits			
SPrAcc	0	Shifting in a zero value requests completion of the Fastdata access. The PrAcc bit in the EJTAG Control register is overwritten with zero when the access succeeds. (The access succeeds if PrAcc is one and the operation address is in the legal dmseg Fastdata area.) When successful, a one is shifted out. Shifting out a zero indicates a Fastdata access failure. Shifting in a one does not complete the Fastdata access and the PrAcc bit is unchanged. Shifting out a one indicates that the access would have been successful if allowed to complete and a zero indicates the access would not have successfully completed.	R/W	Undefined

The FASTDATA access is used for efficient block transfers between dmseg (on the probe) and target memory (on the processor). An “upload” is defined as a sequence of processor loads from target memory and stores to dmseg. A “download” is a sequence of processor loads from dmseg and stores to target memory. The “Fastdata area” specifies the legal range of dmseg addresses (0xFFFF FFFF FF20 0000 - 0xFFFF FFFF FF20 000F) that can be used for uploads and downloads. The Data + Fastdata registers (selected with the FASTDATA instruction) allow efficient completion of pending Fastdata area accesses.

During Fastdata uploads and downloads, the processor will stall on accesses to the Fastdata area. The PrAcc (processor access pending bit) will be 1 indicating the probe is required to complete the access. Both upload and download accesses are attempted by shifting in a zero SPrAcc value (to request access completion) and shifting out SPrAcc to see if the attempt will be successful (i.e., there was an access pending and a legal Fastdata area address was used). Downloads will also shift in the data to be used to satisfy the load from dmseg’s Fastdata area, while uploads will shift out the data being stored to dmseg’s Fastdata area.

As noted above, two conditions must be true for the Fastdata access to succeed. These are:

- PrAcc must be 1, i.e., there must be a pending processor access.
- The Fastdata operation must use a valid Fastdata area address in dmseg (0xFFFF FFFF FF20 0000 to 0xFFFF FFFF FF20 000F).

Table 10-40 shows the values of the PrAcc and SPrAcc bits and the results of a Fastdata access.

Table 10-40 Operation of the FASTDATA access

Probe Operation	Address Match check	PrAcc in the Control Register	LSB (SPrAcc) shifted in	Action in the Data Register	PrAcc changes to	LSB shifted out	Data shifted out
Download using FASTDATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	write data	0 (SPrAcc)	1	valid (previous) data
		0	x	none	unchanged	0	invalid
Upload using FASTDATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	read data	0 (SPrAcc)	1	valid data
		0	x	none	unchanged	0	invalid

There is no restriction on the contents of the Data register. It is expected that the transfer size is negotiated between the download/upload transfer code and the probe software. Note that the most efficient transfer sizes are double-word for 64-bit processors.

The Rocc bit of the Control register is not used for the FASTDATA operation.

10.5.2.7 Bypass Register (TAP Instruction BYPASS, EJTAGBOOT, NORMALBOOT, or Unused)

The Bypass register is a one-bit read-only register, which provides a minimum shift path through the TAP. This register is also defined in IEEE 1149.1. Figure 10-24 shows the format of the Bypass register; Table 10-41 describes the Bypass register field.

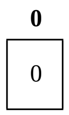


Figure 10-24 Bypass Register Format

Table 10-41 Bypass Register Field Description

Fields		Description	Read/ Write	Power-up State
Name	Bits			
0	0	Ignored on writes; returns zero on reads.	R	0

10.5.3 Example of EJTAG Memory Access through Processor Access

The processor access feature makes it possible for the probe to handle accesses from the processors to the specific EJTAG memory area (dmseg). Thus the processor can execute a debug handler from EJTAG memory, whereby applications that are not prepared with EJTAG code in the system memory still can be debugged.

The probe can get information about the access through the TAP as shown in Table 10-42.

Table 10-42 Information Provided to Probe at Processor Access

Information	Field and Register
Pending processor access	PrAcc field in the EJTAG Control register
Read or write access	PRnW field in the EJTAG Control register
Size and data location	Psz field in EJTAG Control register, and two or three LSBs in the Address register
Address	Address register
Data	Data register

The servicing of processor accesses works with a polling scheme, where the PrAcc bit is polled until a pending processor access is indicated by PrAcc equal to 1. Then the Address register is read to get the address of the transaction, and the Data register is accessed to get the write data or provide the read data. Finally the PrAcc bit is cleared, in order to finish the access from the processor.

In addition, the ProbTrap and ProbEn bits control the debug exception vector location and the indication to the processor that the probe will service accesses to the EJTAG memory through processor accesses.

Handling of processor access in relation to reset requires specific handling. A pending processor access is cleared at reset. At the same time, the Rocc bit is set, thereby inhibiting any processor accesses to be finished until Rocc is cleared. Thus the probe will have to acknowledge that a reset occurred, and will thereby not accidentally finish a processor access due to a processor access that occurred before the reset.

The following subsections show examples of servicing read and write processor accesses.

10.5.3.1 Write Processor Access

Figure 10-25 shows a possible flow for servicing a write processor access. A halfword store is performed to address 0xF FF20 1232 when running little-endian with the value 0x5678.

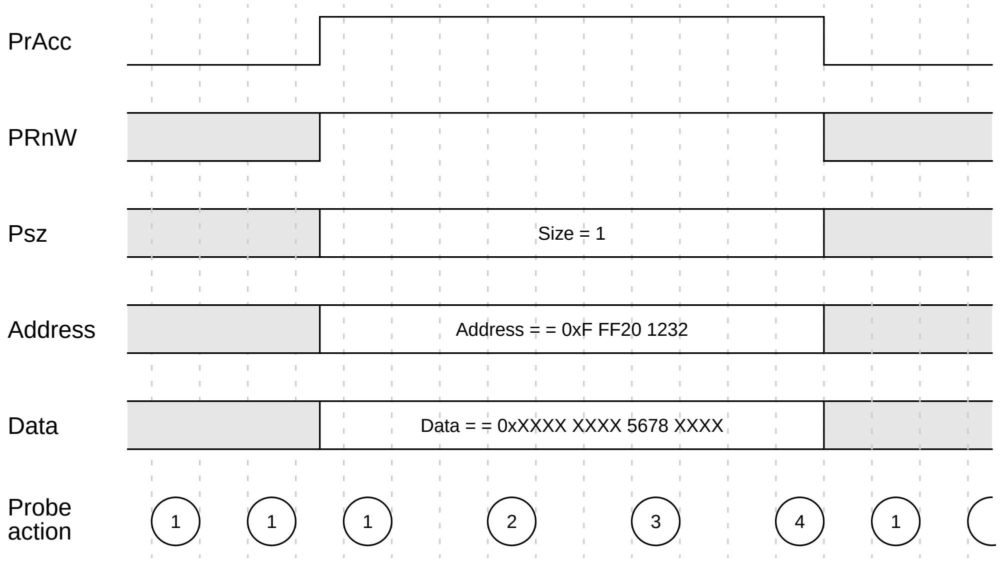


Figure 10-25 Write Processor Access Example

The different probe actions shown on the figure are described below:

1. The EJTAG Control register is polled to get the indication for a pending PrAcc bit. The PrAcc bit is written to 1 when polling, in order to prevent a processor access from finishing before being serviced. The values of PRnW and Psz are saved when PrAcc indicates a pending processor access.
2. The Address register is read. It contains the address of the store resulting in the write processor access.
3. The Data register is read, which contains the data from the store resulting in the write processor access.
4. The PrAcc bit is written to 0, in order to finish the processor access.

The probe must update the appropriate bytes in its internal memory used for EJTAG memory with the value of the write.

Notice that the two lower and the four upper bytes of the Data register are undefined, and that the two lower bytes of the saved register are shifted up two bytes in the Data register as on a data bus for an external memory system. The Address register in this case contains the address from the store; however, for some accesses, this is not the case because the three LSBs are modified for some accesses depending on size and address.

10.5.3.2 Read Processor Access

Figure 10-26 shows a possible flow for servicing a read processor access. Figure 10-26 shows a doubleword load/fetch from address 0xFFFF FFFF FF20 3450.

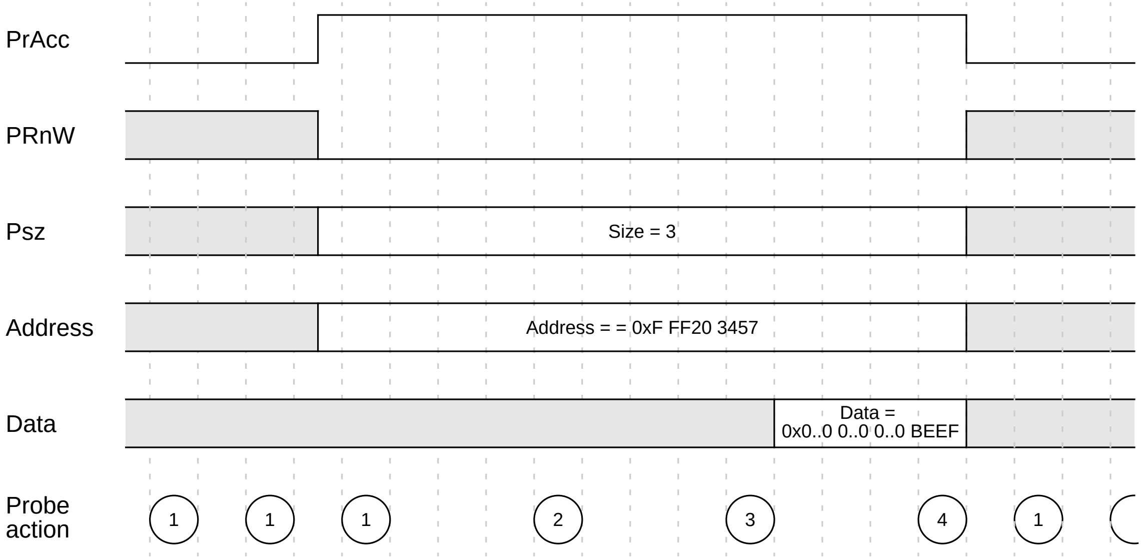


Figure 10-26 Read Processor Access Example

The different probe actions shown in the figure are described below:

1. The EJTAG Control register is polled to get the indication for a pending PrAcc bit. The PrAcc bit is written to 1 when polling, in order to prevent a processor access from finishing before being serviced. The values of PRnW and Psz are saved when PrAcc indicates a pending processor access.
2. The Address register is read. It contains the address of the load/fetch resulting in the write processor access, with the three LSBs modified to allow size indication together with the Psz.
3. The Data register is written with the data to return for the load/fetch, resulting in the read processor access.
4. The PrAcc bit is cleared, in order to finish the processor access.

The probe must provide data for the read processor access from the internal EJTAG memory.

Notice that the Address register does not contain the direct address from the access, because the three LSBs are modified to indicate the size in conjunction with Psz. Also notice that in this case, there is no shifting of the data returned for the processor access by writing to the Data register, because a doubleword is provided. For other accesses, the Data register must be written with a shifted value depending on the specific access.

Chapter 11

Instruction Set Overview

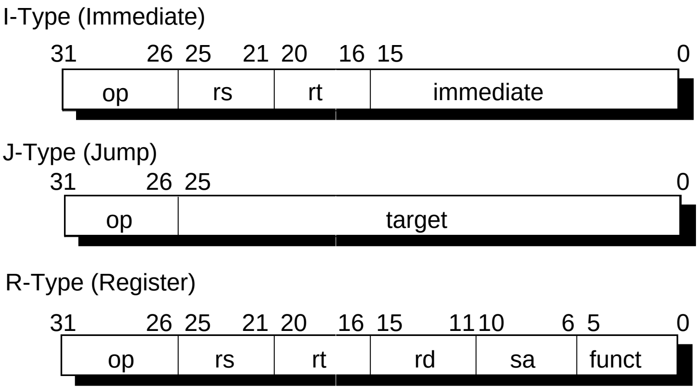
This chapter provides an overview of the 5K microprocessor core instruction set, including instruction formats and basic instruction types. Refer to Chapter 12, “Instructions,” for a detailed description of each instruction.

This chapter contains the following sections:

11.1 CPU Instruction Formats

A 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*). The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complex (and less-frequently used) operations and addressing modes from these three formats, as needed. The three instruction formats are shown in Figure 11-1.



op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination) register specifier 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 11-1 CPU Instruction Formats

11.2 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*.

11.2.1 Scheduling a Load Delay Slot

A load instruction that does not allow its result to be used by the instruction that immediately follows it is called a *delayed load instruction*. The instruction slot immediately following the delayed load instruction is referred to as the *load delay slot*.

In the 5K core processor, the instruction immediately following a load instruction can use the contents of the loaded register; in such cases, hardware interlocks insert additional real cycles. Though not required, the scheduling of load delay slots can be desirable, both for performance and R-Series processor compatibility. For more information about load scheduling, refer to Section 2.7, "Instruction Scheduling" on page 16.

11.2.2 Access Types

Access type refers to the size of a data item to be loaded or stored. The access type is specified in the opcode of the load or store instruction, and, for some instructions, by the three low-order address bits (for example, LWL and LWR).

Regardless of access type or byte ordering (endianess), the given address always 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 11-1. Only the combinations shown in Table 11-1 are permitted; other combinations cause Address Error exceptions.

Table 11-1 Byte Access Within a Doubleword

Access Type	Low-Order Address Bits			Bytes Accessed															
				Big Endian (63----------------31-------------------0)								Little Endian (63----------------31-------------------0)							
	2	1	0	Byte								Byte							
Doubleword	0	0	0	0	1	2	3	4	5	6	7	7	6	5	4	3	2	1	0
Septibyte	0	0	0	0	1	2	3	4	5	6			6	5	4	3	2	1	0
	0	0	1		1	2	3	4	5	6	7	7	6	5	4	3	2	1	
Sextibyte	0	0	0	0	1	2	3	4	5					5	4	3	2	1	0
	0	1	0			2	3	4	5	6	7	7	6	5	4	3	2		
Quintibyte	0	0	0	0	1	2	3	4							4	3	2	1	0
	0	1	1				3	4	5	6	7	7	6	5	4	3			
Word	0	0	0	0	1	2	3									3	2	1	0
	1	0	0					4	5	6	7	7	6	5	4				
Triplebyte	0	0	0	0	1	2											2	1	0
	0	0	1		1	2	3									3	2	1	
	1	0	0					4	5	6			6	5	4				
	1	0	1						5	6	7	7	6	5					
Halfword	0	0	0	0	1													1	0
	0	1	0			2	3									3	2		
	1	0	0					4	5					5	4				
	1	1	0							6	7	7	6						
Byte	0	0	0	0															0
	0	0	1		1													1	
	0	1	0			2											2		
	0	1	1				3									3			
	1	0	0					4							4				
	1	0	1						5					5					
	1	1	0							6			6						
	1	1	1								7	7							

11.3 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 can be grouped into the following four categories:

- ALU Immediate instructions
- Three-operand Register-type Instructions
- Shift Instructions
- Multiply And Divide Instructions

11.3.1 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 MULT (integer multiply) instruction is saved in the HI and LO registers. If a multiply instruction is followed by an MFHI or MFLO before the product is available, the pipeline interlocks until the product becomes available. For information on the latency and repeat rates for integer multiply and divide operations, refer to Table 2-3 on page 19 and Table 2-4 on page 20.

11.4 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 (the instruction occupying the *delay slot*) always executes while the target instruction is being fetched from storage.

11.4.1 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 the J-type format, the 26-bit target address is shifted left by 2 bits and combined 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 use the 32-bit or 64-bit byte address contained in one of the general purpose registers.

11.4.2 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.

11.5 Control Instructions

Control instructions allow the software to initiate traps. They are always R-type.

11.6 Coprocessor Instructions

Coprocessor instructions perform operations in the Coprocessors that are supported by the 5K architecture. Coprocessor loads and stores are I-type. Coprocessor computational instructions have coprocessor-dependent formats. These instructions are described in Chapter 12, “Instructions.”

11.7 Enhancements to the MIPS Architecture

The 5K core execution unit implements the MIPS64 ISA[1], which includes the following instructions:

- CLO: Count Leading Ones
- DCLO: Double Count Leading Ones
- CLZ: Count Leading Zeros
- DCLZ: Double Count Leading Zeros
- MADD: Multiply and Add Word
- MADDU: Multiply and Add Unsigned Word
- MSUB: Multiply and Subtract Word
- MSUBU: Multiply and Subtract Unsigned Word
- MUL: Multiply Word to Register

11.7.1 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.

11.7.2 DCLO - Double Count Leading Ones

The DCLO instruction counts the number of leading ones in a doubleword. The 64-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 64 bits are set in the GPR *rs*, the result written to the GPR *rd* is 64.

11.7.3 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.

[1] Refer to the MIPS64 Specification, rev 1.0 or later.

11.7.4 DCLZ - Double Count Leading Zeros

The DCLZ instruction counts the number of leading zeros in a doubleword. The 64-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 64 bits are cleared in the GPR *rs*, the result written to the GPR *rd* is 64.

11.7.5 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.

Note that this instruction does not provide the capability or writing directly to the target GPR. This is to prevent having two destination registers, which would be difficult to support in potential high-performance processor implementations that rename registers.

11.7.6 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.

Note that this instruction does not provide the capability or writing directly to the target GPR. This is to prevent having two destination registers, which would be difficult to support in potential high-performance processor implementations that rename registers.

11.7.7 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.

Note that this instruction does not provide the capability or writing directly to the target GPR. This is to prevent having two destination registers, which would be difficult to support in potential high-performance processor implementations that rename registers.

11.7.8 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.

Note that this instruction does not provide the capability of writing directly to the target GPR. This is to prevent having two destination registers, which would be difficult to support in potential high-performance processor implementations that rename registers.

11.7.9 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.

Chapter 12

Instructions

This chapter provides a detailed guide to the 5K core instruction set, which is compliant with the MIPS64 architecture. It contains the following sections:

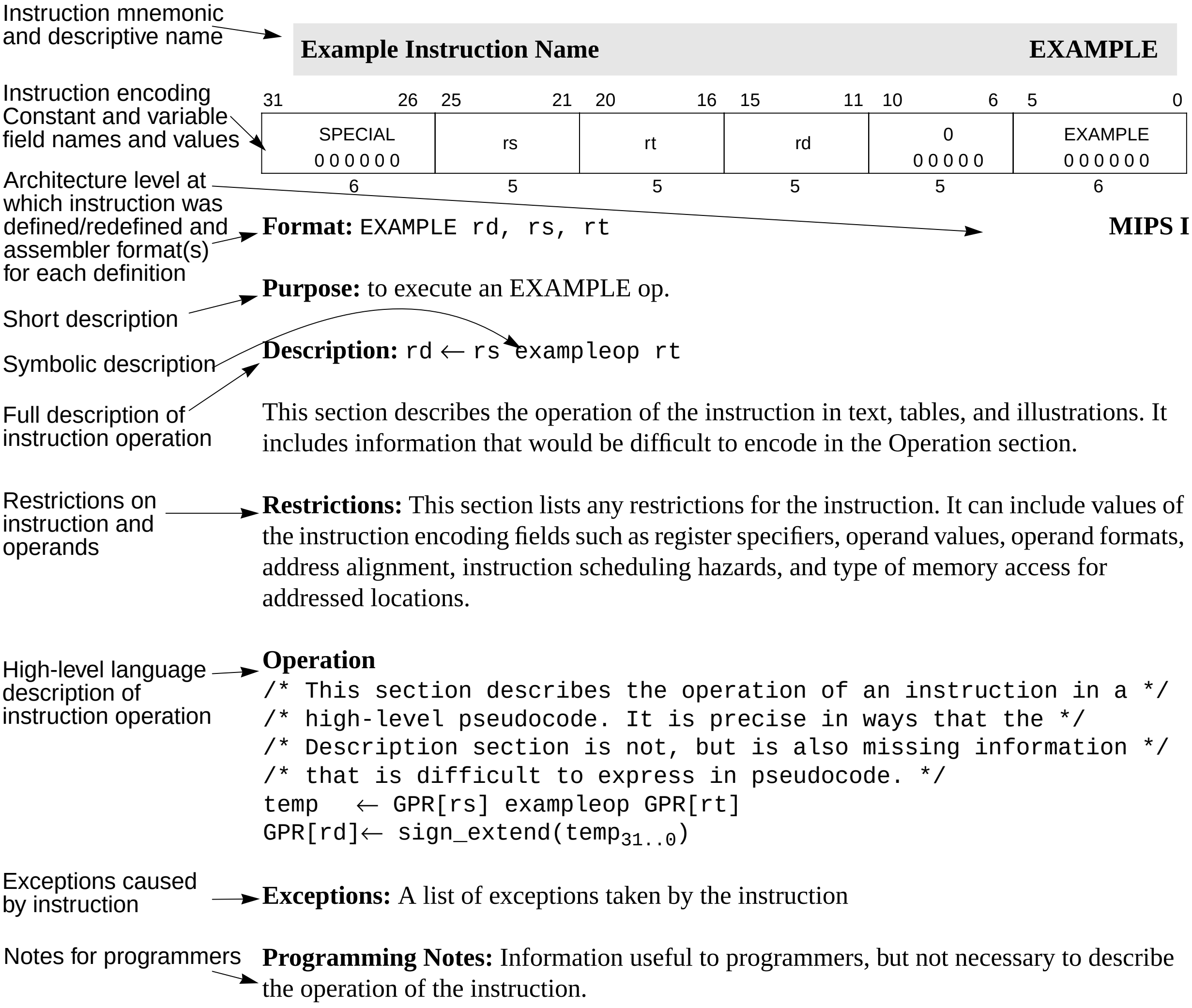
- Section 12.1, "Example Instruction Page"
- Section 12.2, "Coprocessor 0 (CP0) Hazards"
- Section 12.3, "Instruction Summary"
- Section 12.4, "Instruction Bit Encodings"
- Section 12.5, "Instruction Set"

12.1 Example Instruction Page

Figure 12-1 shows an annotated example of an instruction page. Each instruction page includes some or all of the following fields discussed in the associated subsections:

- Section 12.1.1, "Instruction Descriptive Name and Mnemonic"
- Section 12.1.2, "Instruction Fields"
- Section 12.1.3, "Format Field"
- Section 12.1.4, "Purpose Field"
- Section 12.1.5, "Description Field"
- Section 12.1.6, "Restrictions Field"
- Section 12.1.7, "Operation Field"
- Section 12.1.8, "Exceptions Field"

The annotated instruction page in Figure 12-1 contains a brief description of each field. Each field is also described in detail in subsections following the figure.


Instruction mnemonic and descriptive name
Instruction encoding Constant and variable field names and values
Architecture level at which instruction was defined/redefined and assembler format(s) for each definition
Short description
Symbolic description
Full description of instruction operation
Restrictions on instruction and operands
High-level language description of instruction operation
Exceptions caused by instruction
Notes for programmers

Example Instruction Name **EXAMPLE**

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

Format: `EXAMPLE rd, rs, rt` **MIPS I**

Purpose: to execute an EXAMPLE op.

Description: `rd ← rs exampleop rt`

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: This section lists any restrictions for the instruction. It 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.

Operation

```
/* This section describes the operation of an instruction in a */
/* high-level pseudocode. It is precise in ways that the */
/* Description section is not, but is also missing information */
/* that is difficult to express in pseudocode. */
temp   ← GPR[rs] exampleop GPR[rt]
GPR[rd]← sign_extend(temp31..0)
```

Exceptions: A list of exceptions taken by the instruction

Programming Notes: Information useful to programmers, but not necessary to describe the operation of the instruction.

Figure 12-1 Example Instruction Description

12.1.1 Instruction Descriptive Name and Mnemonic

The instruction's descriptive name and mnemonic are printed as page headings for each instruction, as shown below.

Add Word **ADD**

12.1.2 Instruction Fields

The instruction fields that encode the instruction word in register format are shown directly below the instruction's descriptive name and mnemonic. Field descriptions use the following conventions:

- Opcode names and the values of constant fields are shown in upper case (for example, SPECIAL and ADD in Figure 12-2).

- Variable fields are shown with the lowercase names as used in the instruction description (for example, *rs*, *rt* and *rd* in Figure 12-2).
- Unnamed fields containing all zeros are fields that are currently unused and are required to be zero (for example, bits 10:6 in Figure 12-2).

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

Figure 12-2 Example of Instruction Fields

12.1.3 Format Field

The *Format* field shows the assembler formats for the instruction and the architecture level at which the instruction was originally defined. If the instruction definition has been 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 example, see the BC2F instruction). The MIPS architecture levels are inclusive; higher architecture levels include all instructions in previous levels. Extensions to instructions are also backwards compatible, that is, the original assembler formats are valid for the extended architecture.

Format: `ADD rd, rs, rt` **MIPS I**

In the assembler formats, literals are shown in upper case, and variables (operand names) are shown in lower case. The architectural level at which the instruction was first defined, for example "MIPS I," is shown to the right. In some cases, there is more than one assembler format for each architecture level.

Format lines sometimes include parenthetical comments to help explain variations in the formats (see the BC2F instruction). These comments are not part of the assembler format.

12.1.4 Purpose Field

The *Purpose* field provides a short description of the instruction's use.

Purpose: to add 32-bit integers. If overflow occurs, then trap.

12.1.5 Description Field

In the *Description* field, a one-line symbolic description of the instruction, if feasible, appears immediately to the right of the Description heading. Its 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 Description field describes 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.

12.1.6 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 JALR)
- Alignment requirements for memory addresses (for example, see LW)
- Valid values of operands (for example, see DADD)
- Order of instructions necessary to guarantee correct execution. These ordering constraints avoid pipeline hazards for which some processors do not have hardware interlocks (for example, see MUL).
- Valid memory access types (for example, see LL and SC)

12.1.7 Operation Field

The *Operation* field describes the operation of the instruction in pseudocode that uses 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]    ←  $GPR[rt]_{31..0}$
```

12.1.8 Exceptions Field

The *Exceptions* field lists the exceptions that can be caused by the *Operation* of the instruction. Exceptions that can be caused by the instruction fetch (for example, TLB Refill) are omitted, as well as exceptions that can be caused by asynchronous external events, such as an interrupt.

Exceptions:

Integer Overflow

12.2 Coprocessor 0 (CP0) Hazards

The state of the system coprocessor (CP0)—namely, the contents of registers and the TLB— affects the operation of various pipeline stages of a MIPS CPU, and manipulation of this state may produce results that are not detectable by a number of subsequent instructions. While in theory it may be possible to fully interlock the CP0 state and conceal the pipeline structure, this is not done. The effect of an instruction on CP0 state is, in general, not detectable by or applicable to the next instruction to be issued. The delay from a change of CP0 state until the new state is available for use by subsequent instructions is known as a *CP0 hazard*. It is the responsibility of system programmers and programming tools to ensure that CP0 hazards do not result in incorrect system operation.

Note: This chapter describes the CP0 hazards for the 5K processor. However, MIPS Technologies Inc. reserves the right to make future changes in the processor which may affect the size of any of the hazards. Software intended to be run on other MIPS processors should therefore also take into account the CP0 hazards of those processors. Also note that the 5K processor may issue more than one instruction per cycle. For a MIPS implementation in which the number of instructions issued per cycle may be greater than one, the duration of an instruction hazard may be greater than the duration in stages as calculated in this chapter. It is for this reason that the MIPS64 architecture defines the SSNOP instruction which forces the sequential processing of NOPs in a multi-issue design. If software whishes to fill an instruction hazard with other instructions than SSNOPs, possible multi-issue should be taken into account on the 5K processor if one or more of these instructions are FPU or COP2 instructions. Programmers who whish that there code is portable between different implementations of the MIPS architecture should not base their work on the 5K specific documentation given in this manual, but rather on "MIPS64^{TM} Architecture for Programmers Volume III: The MIPS64^{TM} Previleged Resource Architecture" which is freely available on http://www.mips.com.

To simplify the description of hazard conditions and calculations, the 6 stages of the pipeline—I, D, R, E, M, and W—are numbered 0 - 5. (The pipeline is described in Chapter 2, “Pipeline.”) In addition, for a few instructions that perform some of their operations in cycles following the W stage, numbers greater than 5 are used. When the processor is running in cached memory space, the instruction fetch unit (IFU) speculatively fetches instructions before they are needed by the pipeline. It is therefor possible for a number of instructions to have been already fetched and ready in the IFU buffer before they enter the D stage, and thus there may be a number of instructions in the I stage. For reasons of clarity, the oldest instruction in the I stage is said to be in stage 0, and younger instructions are given negative numbers. The size of the negative numbers does not reflect the amount of speculation done, but rather reflects what is needed to calculate the correct CP0 hazards, as explained below. The CP0 takes steps to reduce the amount of speculation done by the IFU around instructions creating CP0 hazards, in order to limit the size of these hazards.

Table 12-1 describes when the modified CP0 state becomes available for use by other instructions (the “available stage”) and when the CP0 state is actually used for different operations (the “used stage”). The instruction that modifies CP0 state is called the *writer* instruction, and the instruction using the CP0 state is called the *user* instruction. Instructions held in the IFU buffer prior to the I stage are denoted by negative numbers.

The following steps are used to determine a hazard delay:

1. Find the pipeline stage of the writer instruction in which the new CP0 state is available. For example, the MTC0 instruction writes a CP0 register in stage 5, and the new value is available in stage 6.
2. Find the pipeline stage in which the user instruction uses the CP0 state changed by the writer instruction. The instruction fetch uses $Status_{UX/KX}$ as early as stage 1 to determine the addressing mode for the fetch. Load/store instructions use the same information in stage 4 for the data reference.
3. Calculate the number of instructions that must be inserted between the hazardous pair, using the formula: *available_stage[writer] - (use_stage[user] + 1*). For example, for an MTC0 that changes $Status_{UX/KX}$ and an instruction fetch using the new register contents, with the MTC0 data available at stage 6 and the instruction fetch using the new value as early as stage 1, the computation is: 6 – (-1 + 1) = 6. This means that 6 instructions must be inserted between the MTC0 and the first instruction fetch using the new mode. If the result of the computation is less than or equal to zero, there is no hazard, and no instructions are required between the pair. If the result is greater than zero, instructions (possibly NOPs) guaranteed to consume the resulting number of cycles must be added between them.

Note: Stalls may be inserted between pipeline stages because of memory system performance, etc. For this reason, software shall NOT depend on the exact size of a hazard, but regard the value obtained from the above calculation as an upper limit for the hazard. For example, it would be ill-advised to modify a TLB contents (using a TLBWR instruction) in a program that is executing in an address space that is mapped by the TLB entry which may be replaced by the new information in the TLBWR instruction.

In short, to identify a hazard, look for a writer/user pair of specific CP0 state, and use the available/used stage information in Table 12-1 to calculate the delay required between the write and user. If no delay is required, there is no hazard. If there is a hazard, enough instructions must be placed between the writer and user so that the written information is available when the user needs it. Notice that future versions of the 5K processor may issue more than one instruction per cycle and thus require the use of the SSNOP instruction to ensure that software does not violate the CP0 hazard.

Note: Instructions that are inserted between a writer/user hazardous pair must NOT depend on or modify the CP0 state covered by the hazard. NOP and SSNOP instructions may always be used to separate the writing and using instructions.

Table 12-1 5K CP0 Hazard Description Table

Instruction or Event	CP0 Data Written; Latest Stage Available		CP0 Data Used; Earliest Stage Used	
MTC0 / DMTC0	CPR[0,rd]	6^{δ}		
MFC0 / DMFC0			CPR[0,rd]	4^{ϕ}
TLBR	*PageMask*, *EntryHi*, *EntryLo0*, *EntryLo1*	6	*Index*, TLB Entry	4
TLBWI TLBWR	TLB entry	5^{γ}	*Index* or *Random*, *PageMask*, *EntryHi*, *EntryLo0*, *EntryLo1*	4-7
TLBP	Index	6	*PageMask*, *EntryHi*, TLB entry	4
ERET	*Status*$_{[EXL, ERL]}$	5^{α}	*EPC* or *ErrorEPC*	3^{α}
	LLbit	5		
CACHE; Index Load Tag operation	*TagLo*, *TagHi*, *ErrCtl*	5^{ε}		
CACHE; Index Store Tag operation			*TagLo*, *TagHi*, *ErrCtl*	4^{ε}
CACHE all operations	Cache line (see note)	5^{ε}	cache line (see note)	4^{ε}
Load/Store			*EntryHi*$_{ASID}$, *Status*$_{[KSU, EXL, ERL, RE,KX, UX]}$, *Config*$_{[K0, KU,K23]}$, TLB entry	4
Watchpoint on read/write			*WatchHi*, *WatchLo*	4
Load/Store exception	*EPC*, *Status*, *Cause*, *BadVaddr*, *Context*, *XContext*	6		
Instruction fetch exception	*EPC*, *Status*	6		
	Cause, *BadVAddr*, *Context*, *XContext*	6		
Instruction fetch			*EntryHi*$_{ASID,}$ *Status*$_{[KSU, EXL, ERL, RE,KX,UX]}$, *Config*$_{[K0, KU,K23]}$	$-1^{\alpha\beta}$
			TLB entry (mapped addresses)	-1^{β}
Watchpoint on instruction fetch			*WatchHi*, *WatchLo*	2
Coprocessor 0 usable test			*Status*$_{[CU0, KSU, EXL, ERL]}$	2
Coprocessor 1 and 2 usable test			*Status*$_{[CU1,CU2,MX]}$	1
Dual issue disable; FPU register mode test$^{\pi}$ and test of *Config*$_{DID}$			*Status*$_{FR,}$ *Config*$_{DID}$	0

Table 12-1 5K CP0 Hazard Description Table (Continued)

Instruction or Event	CP0 Data Written; Latest Stage Available		CP0 Data Used; Earliest Stage Used	
Interrupt signals sampled[ρ]			*Cause*$_{IP}$	4
			Status$_{[IM, IE, EXL, ERL]}$ disabling interrupts	4
			Status$_{[IM, IE, EXL, ERL]}$ enabling interrupts through MTC0	1
TLB shutdown	*Status*$_{TS}$	6		

EntryHi$_{ASID}$ refers to the ASID field of the *EntryHi* register, *Config*$_{[K0,KU,K23]}$ refers to the K0, KU, and K23 fields of the *Config* register, etc.

α The ERET is interlocked to ensure that the fetch of the return instruction sees the correct CP0 state. The fetch of the return instruction is started following stage 3 of the ERET instruction. In addition, any change in CP0 state leading up to the ERET is also interlocked by the hardware except for state changes that sets the CP0 unusable. For optimal performance, it is recommended that software observe the underlying hardware hazards since the interlock mechanism has not been optimized for cycle performance in all cases.

β Instructions are fetched speculatively by the processor. However, to reduce CP0 hazards, the amount of speculation is limited by the CP0 around any hazard-creating instructions.

δ With an MTC0 to *Status* that modifies KSU and sets EXL or ERL, it is possible for the six instructions following the MTC0 to be executed incorrectly in the new mode and incorrectly in Kernel Mode. This can be avoided by setting EXL first, and only later changing the value of KSU.

ε There must be two non-load, non-CACHE instructions between a store and a CACHE instruction that is directed to the same primary cache line as the store.

γ The TLBWI/TLBWR instructions write the new TLB entries in the three cycles following their M stage. However, the CP0 interlocks the TLB write instructions and operations using the TLB, so that the new TLB entries are available as indicated in this table.

ρ Interrupts are recognized in stages 2 - 4 (the R, E, and M stages). To provide compatibility with other MIPS processors, the effect of an MTC0, which modifies *Status*$_{[IM, IE, EXL, ERL]}$ in order to enable an interrupt, is first visible to the fourth instruction following the MTC0; that is, the three instructions following the MTC0 will be executed with interrupts enabled according to the old value of *Status*$_{[IM, IE, EXL, ERL]}$. Disabling of interrupts using an MTC0 instruction takes effect immediately following the W stage of the MTC0 instruction.

ϕ MTC0 directly followed by an MFC0 from the same register is fully interlocked—the second MFC0 will stall in order to guarantee that it will return the new data written by the MTC0. Note, however, that the *DataHi* register is aliased with the upper 32 bits of *DataLo*, and that this alias is NOT interlocked.

π The FR bit is used as earliest as the I stage to disable dual issue in some cases in 32 bit register mode. See Section 2.4, "Limited Dual Issue" for further details.

Table 12-2 lists some hazard and non-hazard conditions, and the number of instructions required between the writer and the user. The table shows the operation that creates the hazard and the calculation for the required number of intervening instructions.

Table 12-2 5K CP0 Hazards and Calculated Delay Times

Writer	→	User	Hazard On	Instructions Between	Calculation
MTC0	→	MFC0	CP0 register	0	interlocked by hardware

Table 12-2 5K CP0 Hazards and Calculated Delay Times (Continued)

Writer	→	User	Hazard On	Instructions Between	Calculation
TLBWR/TLBWI	→	TLBP, TPBR	TLB entry	0	5-(4+1)
		load/store using new TLB entry	TLB entry	0	5-(4+1)
		I-fetch using new TLB entry	TLB entry	5	5-(-1+1)
MTCO $\mathit{Status}_{\mathrm{CU0}}$	→	Coprocessor instruction needs CU0 set	$\mathit{Status}_{\mathrm{CU0}}$	3	6-(2+1)
MTCO $\mathit{Status}_{\mathrm{CU1}}$	→	Coprocessor instruction needs CU1 set	$\mathit{Status}_{\mathrm{CU1}}$	4	6-(1+1)
TLBR	→	MFC0 *EntryHi, PageMask*	*EntryHi, PageMask*	1	6-(4+1)
MTC0 *EntryLo0* MTC0 *EntryLo1* MTC0 *EntryHi* MTC0 *PageMask* MTC0 *Index*	→	TLBP TLBR TLBWI TLBWR	*EntryLo0* *EntryLo1* *EntryHi* *PageMask* *Index*	1	6-(4+1)
TLBP	→	MFC0 *Index*	*Index*	1	6-(4+1)
MTC0 *EPC*	→	ERET	*EPC*	0	Interlocked by hardware
MTC0 *Status*	→	ERET	*Status*		
MTC0 $\mathit{Status}_{\mathrm{IE}}$	→	instruction interrupted[a]	$\mathit{Status}_{\mathrm{IE}}$	4	6-(1+1)

[a] You cannot depend on a delay to be in effect if the instruction execution order is changed by exceptions. In this case, for example, the *minimum* delay of IE to be effective is the *maximum* delay before a pending, enabled interrupt can occur.

12.2.1 Hazards on CACHE Instructions Modifying Instruction Cache Contents

When the contents of the instruction cache is updated using a CACHE instruction, the instruction fetch unit does not flush its buffers. Software is therefore required to separate the CACHE instruction and the first instruction fetch from the affected memory locations by an ERET instruction.

12.3 Instruction Summary

12.3.1 Basic Instructions

This section contains detailed descriptions for each 5K core instruction.

Table 12-3 can be used as a quick reference for the 5K core family common instructions.

Table 12-3 5K Core Family Common Instruction Set

Instruction	Description	Function
ADD	Add	`Rd = Rs + Rt`
ADDI	Add Immediate	`Rt = Rs + Immed`
ADDIU	Unsigned Add Immediate	`Rt = (uns)Rs + Immed`

Table 12-3 5K Core Family Common Instruction Set (Continued)

Instruction	Description	Function
ADDU	Unsigned Add	Rd = (uns)Rs + Rt
AND	Logical AND	Rd = Rs & Rt
ANDI	Logical AND Immediate	Rt = Rs & (0_{48} || Immed)
BC1F	Branch On Coprocessor 1 False	if COP1_condition == 0 PC += offset

BC1FL	Branch On Coprocessor 1 False Likely	if COP1_condition == 0
		 PC += offset
		else
		  Ignore Next Instruction

BC1T	Branch On Coprocessor 1 True	if COP1_condition == 1 PC += offset

BC1TL	Branch On Coprocessor 1 True Likely	if COP1_condition == 1
		 PC += offset
		else
		  Ignore Next Instruction

BC2F	Branch On Coprocessor 2 False	if COP2_condition == 0 PC += offset

BC2FL	Branch On Coprocessor 2 False Likely	if COP2_condition == 0
		 PC += offset
		else
		  Ignore Next Instruction

BC2T	Branch On Coprocessor 2 True	if COP2_condition == 1 PC += offset

BC2TL	Branch On Coprocessor 2 True Likely	if COP2_condition == 1
		 PC += offset
		else
		  Ignore Next Instruction

BEQ	Branch On Equal	if Rs == Rt PC += offset

BEQL	Branch On Equal Likely	if Rs == Rt
		  PC += offset
		else
		  Ignore Next Instruction

BGEZ	Branch on Greater Than or Equal To Zero	if !Rs[63]
		  PC += offset

BGEZAL	Branch on Greater Than or Equal To Zero And Link	if !Rs[63]
		  GPR[31] = PC + 8
		   PC += offset

BGEZALL	Branch on Greater Than or Equal To Zero And Link Likely	if !Rs[63]
		  GPR[31] = PC + 8
		   PC += offset
		else
		  Ignore Next Instruction

BGEZL	Branch on Greater Than or Equal To Zero Likely	if !Rs[63]
		  PC += offset
		else
		  Ignore Next Instruction

BGTZ	Branch on Greater Than Zero	if !Rs[63] && Rs != 0
		  PC += offset

BGTZL	Branch on Greater Than Zero Likely	if !Rs[63] && Rs != 0
		  PC += offset
		else
		  Ignore Next Instruction

Table 12-3 5K Core Family Common Instruction Set (Continued)

Instruction	Description	Function

BLEZ	Branch on Less Than or Equal to Zero	if Rs[63] || Rs == 0
		  PC += offset

BLEZL	Branch on Less Than or Equal to Zero Likely	if Rs[63] || Rs == 0
		  PC += offset
		else
		  Ignore Next Instruction

BLTZ	Branch on Less Than Zero	if Rs[63]
		  PC += offset

BLTZAL	Branch on Less Than Zero And Link	if Rs[63]
		  GPR[31] = PC + 8
		   PC += offset

BLTZALL	Branch on Less Than Zero And Link Likely	if Rs[63]
		  GPR[31] = PC + 8
		   PC += offset
		else
		  Ignore Next Instruction

BLTZL	Branch on Less Than Zero Likely	if Rs[63]
		  PC += offset
		else
		  Ignore Next Instruction

BNE	Branch on Not Equal	if Rs != Rt
		  PC += offset

BNEL	Branch on Not Equal Likely	if Rs != Rt
		  PC += offset
		else
		  Ignore Next Instruction

BREAK	Breakpoint	Breakpoint Exception
CACHE	Cache Operation	See Section 12.5, "Instruction Set" on page 233
CFC1	Control From Coprocessor 1	Rt = CCR[1, Rd]
CFC2	Control From Coprocessor 2	Rt = CCR[2, Rd]
CLO	Count Leading Ones	Rd = NumLeadingOnes(Rs[31:0])
CLZ	Count Leading Zeroes	Rd = NumLeadingZeroes(Rs[31:0])
CTC1	Control To Coprocessor 1	CCR[1, Rd] = Rt
CTC2	Control To Coprocessor 2	CCR[2, Rd] = Rt
DADD	Doubleword Add	Rd = Rs + Rt
DADDI	Doubleword Add Immediate	Rt = Rs + Immed
DADDIU	Unsigned Doubleword Add Immediate	Rt = Rs + Immed
DADDU	Unsigned Doubleword Add	Rd = Rs + Rt
DCLO	Doubleword Count Leading Ones	Rd = NumLeadingOnes(Rs)
DCLZ	Doubleword Count Leading Zeros	Rd = NumLeadingZeroes(Rs)

DDIV	Doubleword Divide	LO = Rs / Rt
		HI = Rs % Rt

DDIVU	Unsigned Doubleword Divide	LO = (uns)Rs / Rt
		HI = (uns)Rs % Rt

Table 12-3 5K Core Family Common Instruction Set (Continued)

Instruction	Description	Function
DERET	Debug Exception Return	PC = DEPC Exit Debug Mode

DIV	Divide	LO = Rs / Rt
		HI = Rs % Rt

DIVU	Unsigned Divide	LO = (uns)Rs / Rt
		HI = (uns)Rs % Rt

DMFC0	Doubleword Move From Coprocessor 0	Rt = CPR[0, Rd, sel]
DMFC1	Doubleword Move From Coprocessor 1	Rt = CPR[1, Rd]
DMFC2	Doubleword Move From Coprocessor 2	Rt = CPR[2, Rd]
DMTC0	Doubleword Move To Coprocessor 0	CPR[0, Rd, sel] = Rt
DMTC1	Doubleword Move To Coprocessor 1	CPR[1, Rd] = Rt
DMTC2	Doubleword Move To Coprocessor 2	CPR[2, Rd] = Rt
DMULT	Doubleword Multiply	HI|LO = Rs * Rd
DMULTU	Unsigned Doubleword Multiply	HI|LO = (uns)Rs * Rd
DSLL	Doubleword Shift Left Logical	Rd = Rt << sa
DSLLV	Doubleword Shift Left Logical Variable	Rd = Rt << Rs[4:0]
DSLL32	Doubleword Shift Left Logical Plus 32	Rd = Rt << sa+32
DSRA	Doubleword Shift Right Arithmetic	Rd = Rt >> sa
DSRAV	Doubleword Shift Right Arithmetic Variable	Rd = Rt >> Rs[4:0]
DSRA32	Doubleword Shift Right Arithmetic Plus 32	Rd = Rt >> sa+32
DSRL	Doubleword Shift Right Logical	Rd = (uns)Rt >> sa
DSRLV	Doubleword Shift Right Logical Variable	Rd = (uns)Rt >> Rs[4:0]
DSRL32	Doubleword Shift Right Logical Plus 32	Rd = (uns)Rt >> sa+32
DSUB	Doubleword Subtract	Rd = Rs - Rt
DSUBU	Unsigned Doubleword Subtract	Rd = (uns)Rs - 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[63:28] || offset<<2

JAL	Jump and Link	GPR[31] = PC + 8
		PC = PC[63:28] || 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]

Table 12-3 5K Core Family Common Instruction Set (Continued)

Instruction	Description	Function
LBU	Unsigned Load Byte	Rt = (ubyte)Mem[Rs+offset]
LD	Load Doubleword	Rt = Mem[Rs+offset]
LDC1	Load Doubleword to Coprocessor 1	CPR[1,Rt] = Mem[Rs+offset]
LDC2	Load Doubleword to Coprocessor 2	CPR[2,Rt] = Mem[Rs+offset]
LDL	Load Doubleword Left	See Section 12.5, "Instruction Set" on page 233
LDR	Load Doubleword Right	See Section 12.5, "Instruction Set" on page 233
LDXC1	Load Doubleword Indexed to Coprocessor 1	CPR[1,Rd] = Mem[Rs+Rt]
LH	Load Halfword	Rt = (half)Mem[Rs+offset]
LHU	Unsigned Load Halfword	Rt = (uhalf)Mem[Rs+offset]

LL	Load Linked Word	Rt = (word)Mem[Rs+offset]
		LL = 1
		LLAdr = Rs + offset

LLD	Load Linked Doubleword	Rt = Mem[Rs+offset]
		LL = 1
		LLAdr = Rs + offset

LUI	Load Upper Immediate	Rt = immediate << 16
LUXC1	Load Doubleword Indexed Unaligned to Coprocessor 1	CPR[1,Rd] = Mem[Rs+Rt]
LW	Load Word	Rt = (word)Mem[Rs+offset]
LWC1	Load Word to Coprocessor 1	CPR[1,Rt] = Mem[Rs+offset]
LWC2	Load Word to Coprocessor 2	CPR[2,Rt] = Mem[Rs+offset]
LWL	Load Word Left	See Section 12.5, "Instruction Set" on page 233
LWR	Load Word Right	See Section 12.5, "Instruction Set" on page 233
LWU	Load Word Unsigned	Rt = (uword)Mem[Rs+offset]
LWXC1	Load Word Indexed to Coprocessor 1	CPR[1,Rd] = Mem[Rs+Rt]
MADD	Multiply-Add	HI|LO += Rs * Rt
MADDU	Multiply-Add Unsigned	HI|LO += (uns)Rs * Rt
MFC0	Move From Coprocessor 0	Rt = CPR[0, Rd, sel]
MFC1	Move From Coprocessor 1	Rt = CPR[1, Rd]
MFC2	Move From Coprocessor 2	Rt = CPR[2, Rd]
MFHI	Move From HI	Rd = HI
MFLO	Move From LO	Rd = LO

MOVF	Move Conditional on Floating Point False	if COP1_condition == 0 then
		GPR[rd] = GPR[rs]

MOVN	Move Conditional on Not Zero	if Rt != 0 then
		Rd = Rs

MOVT	Move Conditional on Floating Point True	if COP1_condition == 1 then
		GPR[rd] = GPR[rs]

Table 12-3 5K Core Family Common Instruction Set (Continued)

Instruction	Description	Function

MOVZ	Move Conditional on Zero	`if Rt == 0 then`
		`   Rd = Rs`

MSUB	Multiply-Subtract	`HI|LO -= Rs * Rt`
MSUBU	Multiply-Subtract Unsigned	`HI|LO -= (uns)Rs * Rt`
MTC0	Move To Coprocessor 0	`CPR[0, Rd, sel] = Rt`
MTC1	Move To Coprocessor 1	`CPR[1, Rd] = Rt`
MTC2	Move To Coprocessor 2	`CPR[2, Rd] = Rt`
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 = Rs * Rd`
MULTU	Unsigned Multiply	`HI|LO = (uns)Rs * Rd`
NOR	Logical NOR	`Rd = ~(Rs | Rt)`
OR	Logical OR	`Rd = Rs | Rt`
ORI	Logical OR Immediate	`Rt = Rs | Immed`
PREF	Prefetch	Prefetch data from memory
PREFX	Prefetch Indexed	Prefetch data from memory using (GPR+GPR) addressing
SB	Store Byte	`(byte)Mem[Rs+offset] = Rt`

SC	Store Conditional Word	`if LL == 1`
		`   (word)Mem[Rs+offset] = Rt`
		`Rt = LL`

SCD	Store Condition Doubleword	`if LL == 1`
		`   Mem[Rs+offset] = Rt`
		`Rt = LL`

SD	Store Doubleword	`Mem[Rs+offset] = Rt`
SDBBP	Software Debug Breakpoint	Debug breakpoint exception
SDC1	Store Doubleword from Coprocessor 1	`Mem[Rs+offset] = CPR[1,Rt]`
SDC2	Store Doubleword from Coprocessor 2	`Mem[Rs+offset] = CPR[2,Rt]`
SDL	Store Doubleword Left	See Section 12.5, "Instruction Set" on page 233
SDR	Store Doubleword Right	See Section 12.5, "Instruction Set" on page 233
SDXC1	Store Doubleword Indexed from Coprocessor 1	`Mem[Rs+Rt] = CPR[1,Rd]`
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]`

Table 12-3 5K Core Family Common Instruction Set (Continued)

Instruction	Description	Function

SLT	Set on Less Than	`if Rs < Rt`
		`  Rd = 1`
		`else`
		`  Rd = 0`

SLTI	Set on Less Than Immediate	`if Rs < Immed`
		`  Rt = 1`
		`else`
		`  Rt = 0`

SLTIU	Unsigned Set on Less Than Immediate	`if (uns)Rs < Immed`
		`  Rt = 1`
		`else`
		`  Rt = 0`

SLTU	Unsigned Set on Less Than	`if (uns)Rs < Rt`
		`  Rd = 1`
		`else`
		`  Rd = 0`

SRA	Shift Right Arithmetic	`Rd = Rt >> sa`
SRAV	Shift Right Arithmetic Variable	`Rd = 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	See Section 12.5, "Instruction Set" on page 233
SUB	Subtract	`Rd = Rs - Rt`
SUBU	Unsigned Subtract	`Rd = (uns)Rs - Rt`
SUXC1	Store Doubleword Indexed Unaligned from Coprocessor 1	`Mem[Rs+Rt] = CPR[1,Rd]`
SW	Store Word	`(word)Mem[Rs+offset] = Rt`
SWC1	Store Word from Coprocessor 1	`(word)Mem[Rs+offset] = CPR[1,Rt]`
SWC2	Store Word from Coprocessor 2	`(word)Mem[Rs+offset] = CPR[2,Rt]`
SWL	Store Word Left	See Section 12.5, "Instruction Set" on page 233
SWR	Store Word Right	See Section 12.5, "Instruction Set" on page 233
SWXC1	Store Word Indexed from Coprocessor 1	`(word)Mem[Rs+Rt] = CPR[1,Rd]`
SYNC	Synchronize Memory	See Section 12.5, "Instruction Set" on page 233
SYSCALL	System Call	`SystemCallException`

TEQ	Trap if Equal	`if Rs == Rt`
		`  TrapException`

TEQI	Trap if Equal Immediate	`if Rs == Immed`
		`  TrapException`

TGE	Trap if Greater Than or Equal	`if Rs >= Rt`
		`  TrapException`

TGEI	Trap if Greater Than or Equal Immediate	`if Rs >= Immed`
		`  TrapException`

TGEIU	Unsigned Trap if Greater Than or Equal Immediate	`if (uns)Rs >= Immed`
		`  TrapException`

Table 12-3 5K Core Family Common Instruction Set (Continued)

Instruction	Description	Function

TGEU	Unsigned Trap if Greater Than or Equal	`if (uns)Rs >= Rt`
		`  TrapException`

TLBWI	Write Indexed TLB Entry	See Section 12.5, "Instruction Set" on page 233
TLBWR	Write Random TLB Entry	See Section 12.5, "Instruction Set" on page 233
TLBP	Probe TLB for Matching Entry	See Section 12.5, "Instruction Set" on page 233
TLBR	Read Indexed TLB Entry	See Section 12.5, "Instruction Set" on page 233

TLT	Trap if Less Than	`if Rs < Rt`
		`  TrapException`

TLTI	Trap if Less Than Immediate	`if Rs < Immed`
		`  TrapException`

TLTIU	Unsigned Trap if Less Than Immediate	`if (uns)Rs < Immed`
		`  TrapException`

TLTU	Unsigned Trap if Less Than	`if (uns)Rs < Rt`
		`  TrapException`

TNE	Trap if Not Equal	`if Rs != Rt`
		`  TrapException`

TNEI	Trap if Not Equal Immediate	`if Rs != Immed`
		`  TrapException`

WAIT	Wait for Interrupts	Stall until interrupt occurs
XOR	Exclusive OR	`Rd = Rs ^ Rt`
XORI	Exclusive OR Immediate	`Rt = Rs ^ (uns)Immed`

12.3.2 FPU Instructions

Table 12-4 provides a summary of the floating point instructions implemented only by the 5Kf core.

Table 12-4 5Kf Floating Point Instruction Set

Instruction	Format*	Description	Function
ABS.fmt	S, D	Floating Point Absolute Value	Fd = abs(Fs)
ADD.fmt	S, D	Floating Point Add	Fd = Fs + Ft
C.cond.fmt	S, D	Floating Point Compare	cc[i] = Fs compare_cond Ft
CEIL.L.fmt	S, D	Floating Point Ceiling to Long Fixed Point	Fd = convert_and_round(Fs)
CEIL.W.fmt	S, D	Floating Point Ceiling to Word Fixed Poin	Fd = convert_and_round(Fs)
CVT.D.fmt	S, W, L	Floating Point Convert to Double Floating Point	Fd = convert_and_round(Fs)
CVT.L.fmt	S, D	Floating Point Convert to Long Fixed Point	Fd = convert_and_round(Fs)
CVT.S.fmt	W, D, L	Floating Point Convert to Single Floating Point	Fd = convert_and_round(Fs)
CVT.W.fmt	S, D	Floating Point Convert to Word Fixed Point	Fd = convert_and_round(Fs)
DIV.fmt	S, D	Floating Point Divide	Fd = Fs / Ft
FLOOR.L.fmt	S, D	Floating Point Floor to Long Fixed Point	Fd = convert_and_round(Fs)
FLOOR.W.fmt	S, D	Floating Point Floor to Word Fixed Point	Fd = convert_and_round(Fs)
* Instruction Format Type: S = Single, D = Double, W = Word, L = Longword			

Table 12-4 5Kf Floating Point Instruction Set (Continued)

Instruction	Format*	Description	Function
MADD.fmt	S, D	Floating Point Multiply Add	Fd = Fs * Ft + Fr
MOV.fmt	S, D	Floating Point Move	Fd = Fs
MOVF.fmt	S, D	Floating Point Conditional Move on Floating Point False	if (cc[i] == 0) then Fd = Fs
MOVN.fmt	S, D	Floating Point Conditional Move on Non-Zero	if (Rt != 0) then Fd = Fs
MOVT.fmt	S, D	Floating Point Conditional Move on Floating Point True	if (cc[i] = 1) then Fd = Fs
MOVZ.fmt	S, D	Floating Point Conditional Move on Zero	if (Rt == 0) then Fd = Fs
MSUB.fmt	S, D	Floating Point Multiply Subtract	Fd = Fs * Ft - Fr
MUL.fmt	S, D	Floating Point Multiply	Fd = Fs * Ft
NEG.fmt	S, D	Floating Point Negate	Fd = neg(Fs)
NMADD.fmt	S, D	Floating Point Negative Multiply Add	Fd = neg(Fs * Ft + Fr)
NMSUB.fmt	S, D	Floating Point Negative Multiply Subtract	Fd = neg(Fs * Ft - Fr)
RECIP.fmt	S, D	Floating Point Reciprocal Approximation	Fd = recip(Fs)
ROUND.L.fmt	S, D	Floating Point Round to Long Fixed Point	Fd = convert_and_round(Fs)
ROUND.W.fmt	S, D	Floating Point Round to Word Fixed Point	Fd = convert_and_round(Fs)
RSQRT.fmt	S, D	Floating Point Reciprocal Square Root Approximation	Fd = rsqrt(Fs)
SQRT.fmt	S, D	Floating Point Square Root	Fd = sqrt(Fs)
SUB.fmt	S, D	Floating Point Subtract	Fd = Fs - Ft
TRUNC.L.fmt	S, D	Floating Point Truncate to Long Fixed Point	Fd = convert_and_round(Fs)
TRUNC.W.fmt	S, D	Floating Point Truncate to Word Fixed Point	Fd = convert_and_round(Fs)
* Instruction Format Type: S = Single, D = Double, W = Word, L = Longword			

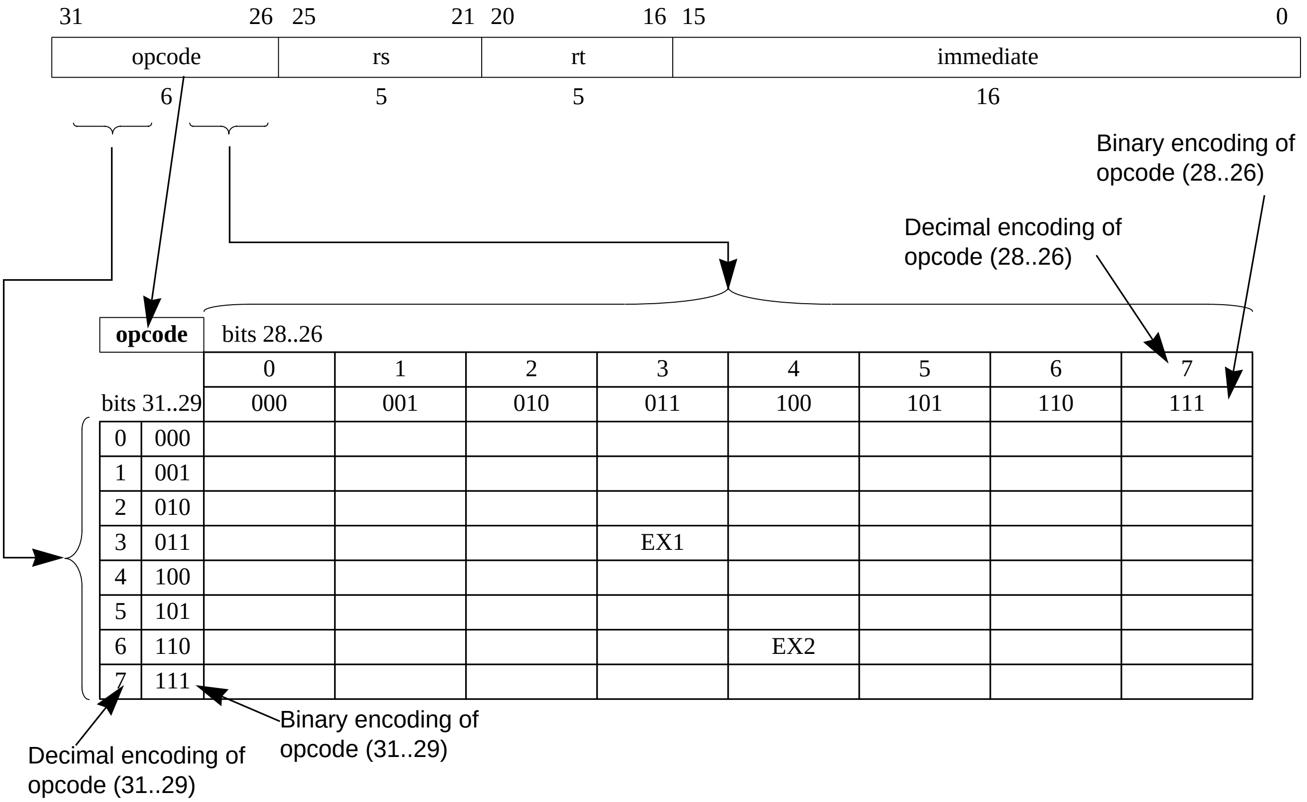
12.4 Instruction Bit Encodings

Instruction encodings are presented in this section. When encoding an instruction, the primary *opcode* field is encoded first. Most *opcode* values completely specify an instruction that has an *immediate* value or offset.

Opcode values that do not specify an instruction instead specify an instruction class. Instructions within a class are further specified by values in other fields. For instance, *opcode* REGIMM specifies the *immediate* instruction class, which includes conditional branch and trap *immediate* instructions.

Figure 12-3 shows a sample encoding table and the instruction *opcode* field this table encodes. Bits 31..29 of the *opcode* field are listed in the leftmost columns of the table. Bits 28..26 of the *opcode* field are listed along the topmost rows of the table. Both decimal and binary values are given, with the first three bits designating the row, and the last three bits designating the column.

An instruction's encoding is found at the intersection of a row (bits 31..29) and column (bits 28..26) value. For instance, the *opcode* value for the instruction labelled EX1 is 33 (decimal, row and column), or 011011 (binary). Similarly, the *opcode* value for EX2 is 64 (decimal), or 110100 (binary).


31
26
25
21
20
16
15
0
opcode
rs
rt
immediate
6
5
5
16
Binary encoding of opcode (28..26)
Decimal encoding of opcode (28..26)
Binary encoding of opcode (31..29)
Decimal encoding of opcode (31..29)

opcode		bits 28..26							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000								
1	001								
2	010								
3	011				EX1				
4	100								
5	101								
6	110					EX2			
7	111								

Figure 12-3 Sample Bit Encoding Table

Tables 12-6 through 12-21 describe the encoding used. Table 12-5 describes the meaning of the symbols used in the tables.

Table 12-5 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction causes a Reserved Instruction exception.
δ	(Also *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
⊥	Operation or field codes marked with this symbol represent instructions which are not legal if the processor is configured to be backward compatible with MIPS32 processors. If the processor is executing in Kernel Mode, Debug Mode, or 64-bit instructions are enabled, execution proceeds normally. In other cases, executing such an instruction causes a Reserved Instruction exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
θ	Operation or field codes marked with this symbol are available to the 5K core user. To avoid multiple conflicting instruction definitions, the partner must notify MIPS Technologies when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction exception for coprocessor instruction encodings for a coprocessor to which access is allowed.
ε	Operation or field codes marked with this symbol are reserved for MIPS Application Specific Extensions. If the ASE is not implemented, executing such an instruction causes a Reserved Instruction exception.

Table 12-5 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
ϕ	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS64 ISA. Software should avoid using these operation or field codes.

Table 12-6 Encoding of the Opcode Field

opcode		*bits 28..26*							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0* δ	*COP1* δ	*COP2* δ	*COP1X* δ⊥	BEQL ϕ	BNEL ϕ	BLEZL ϕ	BGTZL ϕ
3	011	DADDI ⊥	DADDIU ⊥	LDL ⊥	LDR ⊥	*SPECIAL2* δ	JALX ε	*MDMX* εδ	*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU ⊥
5	101	SB	SH	SWL	SW	SDL ⊥	SDR ⊥	SWR	CACHE
6	110	LL	LWC1	LWC2 θ	PREF	LLD ⊥	LDC1	LDC2 θ	LD ⊥
7	111	SC	SWC1	SWC2 θ	*	SCD ⊥	SDC1	SDC2 θ	SD ⊥

Table 12-7 *SPECIAL* Opcode Encoding of Function Field

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL	*MOVCI* δ	SRL	SRA	SLLV	*	SRLV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	DSLLV ⊥	*	DSRLV ⊥	DSRAV ⊥
3	011	MULT	MULTU	DIV	DIVU	DMULT ⊥	DMULTU ⊥	DDIV ⊥	DDIVU ⊥
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	DADD ⊥	DADDU ⊥	DSUB ⊥	DSUBU ⊥
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	DSLL ⊥	*	DSRL ⊥	DSRA ⊥	DSLL32 ⊥	*	DSRL32 ⊥	DSRA32 ⊥

Table 12-8 *REGIMM* Encoding of rt Field

rt		*bits 18..16*							
		0	1	2	3	4	5	6	7
bits 20..19		000	001	010	011	100	101	110	111
0	00	BLTZ	BGEZ	BLTZL ϕ	BGEZL ϕ	*	*	*	*
1	01	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	10	BLTZAL	BGEZAL	BLTZALL ϕ	BGEZALL ϕ	*	*	*	*
3	11	*	*	*	*	*	*	*	*

Table 12-9 *SPECIAL2* Encoding of Function Field

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	*	MSUB	MSUBU	*	*
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	CLZ	CLO	*	*	DCLZ ⊥	DCLO ⊥	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	϶	*	SDBBP

Table 12-10 *MOVCI* Encoding of tf Bit

tf	*bit 16*	
	0	1
	MOVF	MOVT

Table 12-11 *COP0* Encoding of rs Field

rs		*bits 23..21*							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC0	DMFC0 ⊥	*	*	MTC0	DMTC0 ⊥	*	*
1	01	*	*	*	*	*	*	*	*
2	10	*CO* δ							
3	11								

Table 12-12 *COP0* Encoding of Function Field When rs=*CO*

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	TLBR	TLBWI	*	*	*	TLBWR	*
1	001	TLBP	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	ERET	*	*	*	*	*	*	DERET
4	100	WAIT	*	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Table 12-13 *COP1* Encoding of rs Field

rs		*bits 23..21*							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC1	DMFC1 ⊥	CFC1	*	MTC1	DMTC1 ⊥	CTC1	*
1	01	*BC1* δ	*	*	*	*	*	*	*
2	10	*S* δ	*D* δ	*	*	*W* δ	*L* δ⊥	*	*
3	11	*	*	*	*	*	*	*	*

Table 12-14 COP1 Encoding of rt Field When rs=*BC1*

rt	*bits 16*	
bit 17	0	1
0	BC1F	BC1T
1	BC1FL ϕ	BC1TL ϕ

Table 12-15 *COP1* Encoding of Function Field When rs=*S*

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L ⊥	TRUNC.L ⊥	CEIL.L ⊥	FLOOR.L ⊥	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	*MOVCF* δ	MOVZ	MOVN	*	RECIP ⊥	RSQRT ⊥	*
3	011	*	*	*	*	*	*	*	*
4	100	*	CVT.D	*	*	CVT.W	CVT.L ⊥	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7	111	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

Table 12-16 *COP1* Encoding of Function Field When rs=*D*

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L ⊥	TRUNC.L ⊥	CEIL.L ⊥	FLOOR.L ⊥	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	*MOVCF* δ	MOVZ	MOVN	*	RECIP ⊥	RSQRT ⊥	*
3	011	*	*	*	*	*	*	*	*
4	100	CVT.S	*	*	*	CVT.W	CVT.L ⊥	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7	111	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

Table 12-17 *COP1* Encoding of Function Field When rs=*W or L*[a]

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	*	*	*	*	*	*	*
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	CVT.S	CVT.D	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

[a] Format type L is legal only if 64-bit operations are enabled.

Table 12-18 *COP1* Encoding of tf Bit When rs=*S or D,* Function=*MOVCF*

tf	*bit 16*	
	0	1
	MOVF.fmt	MOVT.fmt

Table 12-19 *COP1X* Encoding of Function Field[a]

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	LWXC1	LDXC1	*	*	*	LUXC1	*	*
1	001	SWXC1	SDXC1	*	*	*	SUXC1	*	PREFX
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	MADD.S	MADD.D	*	*	*	*	*	*
5	101	MSUB.S	MSUB.D	*	*	*	*	*	*
6	110	NMADD.S	NMADD.D	*	*	*	*	*	*
7	111	NMSUB.S	NMSUB.D	*	*	*	*	*	*

[a] COP1X instructions are legal only if 64-bit operations are enabled.

Table 12-20 *COP2* Encoding of rs Field

rs		*bits 23..21*							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC2 θ	DMFC2 ⊥ θ	CFC2 θ	θ	MTC2 θ	DMTC2 ⊥ θ	CTC2 θ	θ
1	01	*BC2* δ	θ	θ	θ	θ	θ	θ	θ
2	10	θ	θ	θ	θ	θ	θ	θ	θ
3	11	θ	θ	θ	θ	θ	θ	θ	θ

Table 12-21 COP2 Encoding of rt Field When rs=*BC2*

rt		*bits 18..16*							
		0	1	2	3	4	5	6	7
bits 20..19		000	001	010	011	100	101	110	111
0	00	BC2F θ	BC2T θ	BC2FL φ θ	BC2TL φ θ	θ	θ	θ	θ
1	01	θ	θ	θ	θ	θ	θ	θ	θ
2	10	θ	θ	θ	θ	θ	θ	θ	θ
3	11	θ	θ	θ	θ	θ	θ	θ	θ

12.5 Instruction Set

The following hundreds of pages describes the instruction set.

Floating Point Absolute Value ABS.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ABS 000101
6	5	5	5	5	6

Format: `ABS.S fd, fs` **MIPS32**
`ABS.D fd, fs` **MIPS32**

Purpose:

To compute the absolute value of an FP value

Description: `fd ← abs(fs)`

The absolute value of the value in FPR *fs* is placed in FPR *fd*. The operand and result are values in format *fmt*. *Cause* bits are ORed into the *Flag* bits if no exception is taken.

This operation is arithmetic; a NaN operand signals invalid operation.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*. If they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR(fd, fmt, AbsoluteValue(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

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` **MIPS32**

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 signed-extended and placed into GPR *rd*.

Restrictions:

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

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← (GPR[rs]$_{31}$||GPR[rs]$_{31..0}$) + (GPR[rt]$_{31}$||GPR[rt]$_{31..0}$)
if 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.

Floating Point Add ADD.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	ADD 000000
6	5	5	5	5	6

Format: `ADD.S fd, fs, ft` **MIPS32**
`ADD.D fd, fs, ft` **MIPS32**

Purpose:

To add floating point values

Description: `fd ← fs + ft`

The value in FPR *ft* is added to the value in FPR *fs*. The result is calculated to infinite precision, rounded by using to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. *Cause* bits are ORed into the *Flag* bits if no exception is taken.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*. If they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR (fd, fmt, ValueFPR(fs, fmt) +$_{fmt}$ ValueFPR(ft, fmt))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow

Add Immediate Word ADDI

31 26	25 21	20 16	15 0
ADDI 001000	rs	rt	immediate
6	5	5	16

Format: `ADDI rt, rs, immediate` **MIPS32**

Purpose:

To add a constant to a 32-bit integer. If overflow occurs, then trap**.**

Description: `rt ← 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 sign-extended and placed into GPR *rt*.

Restrictions:

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

Operation:

```
if NotWordValue(GPR[rs]) then
   UNPREDICTABLE
endif
temp ← (GPR[rs]$_{31}$||GPR[rs]$_{31..0}$) + sign_extend(immediate)
if temp$_{32}$ $\neq$ temp$_{31}$ then
   SignalException(IntegerOverflow)
else
   GPR[rt] ← sign_extend(temp$_{31..0}$)
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` **MIPS32**

Purpose:

To add a constant to a 32-bit integer

Description: `rt ← rs + immediate`

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is sign-extended and placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

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

Operation:

```
if NotWordValue(GPR[rs]) then
   UNPREDICTABLE
endif
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` **MIPS32**

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 sign-extended and placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

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

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← 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.

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` **MIPS32**

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 001100	rs	rt	immediate
6	5	5	16

Format: `ANDI rt, rs, immediate` **MIPS32**

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 FP False BC1F

31 26	25 21	20 18	17	16	15 0
COP1 010001	BC 01000	cc	nd 0	tf 0	offset
6	5	3	1	1	16

Format:

```
BC1F   offset (cc = 0 implied)                MIPS32
BC1F   cc, offset                             MIPS32
```

Purpose:

To test an FP condition code and do a PC-relative conditional branch

Description: `if cc = 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 FP condition code bit *CC* is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. An FP condition code is set by the FP compare instruction, C.cond.fmt.

Restrictions:

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

Operation:

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

```
I:      condition ← FPConditionCode(cc) = 0
        target_offset ← $(offset_{15})^{GPRLEN-(16+2)}$ || offset || $0^2$
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

Branch on FP False (cont.) BC1F

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

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 FP False Likely BC1FL

31 26	25 21	20 18	17	16	15 0
COP1 010001	BC 01000	cc	nd 1	tf 0	offset
6	5	3	1	1	16

Format: `BC1FL offset (cc = 0 implied)` **MIPS32**
`BC1FL cc, offset` **MIPS32**

Purpose:

To test an FP condition code and make a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

Description: `if cc = 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 FP *Condition Code* bit *CC* is false (0), the program branches 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.

An FP condition code is set by the FP compare instruction, C.cond.fmt.

Restrictions:

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

Operation:

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

```
I:     condition ← FPConditionCode(cc) = 0
       target_offset ← (offset$_{15}$)$^{GPRLEN-(16+2)}$ || offset || 0$^{2}$
I+1:   if condition then
          PC ← PC + target_offset
       else
          NullifyCurrentInstruction()
       endif
```

Branch on FP False Likely (cont.) BC1FL

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

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.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Branch on FP True **BC1T**

31 26	25 21	20 18	17	16	15 0
COP1 010001	BC 01000	cc	nd 0	tf 1	offset
6	5	3	1	1	16

Format: `BC1T offset (cc = 0 implied)` **MIPS32**
`BC1T cc, offset` **MIPS32**

Purpose:

To test an FP condition code and do a PC-relative conditional branch

Description: `if cc = 1 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 FP condition code bit *CC* is true (1), the program branches to the effective target address after the instruction in the delay slot is executed. An FP condition code is set by the FP compare instruction, C.cond.fmt.

Restrictions:

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

Operation:

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

```
I:     condition ← FPConditionCode(cc) = 1
       target_offset ← $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || $0^2$
I+1:   if condition then
          PC ← PC + target_offset
       endif
```

Branch on FP True (cont.) BC1T

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

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 FP True Likely BC1TL

31 26	25 21	20 18	17	16	15 0
COP1 010001	BC 01000	cc	nd 1	tf 1	offset
6	5	3	1	1	16

Format: `BC1TL offset (cc = 0 implied)` **MIPS32**
`BC1TL cc, offset` **MIPS32**

Purpose:

To test an FP condition code and do a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

Description: `if cc = 1 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 FP *Condition Code* bit *CC* is true (1), the program branches 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.

An FP condition code is set by the FP compare instruction, C.cond.fmt.

Restrictions:

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

Operation:

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

```
I:      condition ← FPConditionCode(cc) = 1
        target_offset ← $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || $0^2$
I+1:    if condition then
           PC ← PC + target_offset
        else
           NullifyCurrentInstruction()
        endif
```

Branch on FP True Likely (cont.) BC1TL

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

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.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Branch on COP2 False BC2F

31 26	25 21	20 18	17	16	15 0
COP2 010010	BC 01000	cc	nd 0	tf 0	offset
6	5	3	1	1	16

Format: `BC2F offset (cc = 0 implied)` **MIPS32**
`BC2F cc, offset` **MIPS32**

Purpose:

To test a COP2 condition code and do a PC-relative conditional branch

Description: `if cc = 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 COP2 condition specified by *CC* is false (0), the program branches to the effective target address after the instruction in the delay slot is executed.

Restrictions:

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

Operation:

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for *tf* and *nd*.

```
I:     condition ← COP2Condition(cc) = 0
       target_offset ← $(offset_{15})^{GPRLEN-(16+2)}$ || offset || $0^2$
I+1:   if condition then
          PC ← PC + target_offset
       endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

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 COP2 False Likely BC2FL

31 26	25 21	20 18	17	16	15 0
COP2	BC	cc	nd	tf	offset
010010	01000		1	0	
6	5	3	1	1	16

Format: `BC2FL offset (cc = 0 implied)` **MIPS32**
`BC2FL cc, offset` **MIPS32**

Purpose:

To test a COP2 condition code and make a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

Description: `if cc = 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 COP2 condition specified by *CC* is false (0), the program branches 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:

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

Operation:

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for *tf* and *nd*.

```
I:      condition ← COP2Condition(cc) = 0
        target_offset ← $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || $0^2$
I+1:    if condition then
           PC ← PC + target_offset
        else
           NullifyCurrentInstruction()
        endif
```

Branch on COP2 False Likely (cont.) BC2FL

Exceptions:

Coprocessor Unusable, Reserved Instruction

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.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Branch on COP2 True BC2T

31 26	25 21	20 18	17	16	15 0
COP2	BC	cc	nd	tf	offset
010010	01000		0	1	
6	5	3	1	1	16

Format: `BC2T offset (cc = 0 implied)` **MIPS32**
`BC2T cc, offset` **MIPS32**

Purpose:

To test a COP2 condition code and do a PC-relative conditional branch

Description: `if cc = 1 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 COP2 condition specified by *CC* is true (1), the program branches to the effective target address after the instruction in the delay slot is executed.

Restrictions:

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

Operation:

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for *tf* and *nd*.

```
I:     condition ← COP2Condition(cc) = 1
       target_offset ← $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || $0^2$
I+1:   if condition then
          PC ← PC + target_offset
       endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

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 COP2 True Likely BC2TL

31 26	25 21	20 18	17	16	15 0
COP2 010010	BC 01000	cc	nd 1	tf 1	offset
6	5	3	1	1	16

Format: `BC2TL offset (cc = 0 implied)` **MIPS32**
`BC2TL cc, offset` **MIPS32**

Purpose:

To test a COP2 condition code and do a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

Description: `if cc = 1 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 COP2 condition specified by *CC* is true (1), the program branches 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:

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

Operation:

This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for *tf* and *nd*.

```
I:      condition ← COP2Condition(cc) = 1
        target_offset ← $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || $0^2$
I+1:    if condition then
           PC ← PC + target_offset
        else
           NullifyCurrentInstruction()
        endif
```

Branch on COP2 True Likely (cont.) BC2TL

Exceptions:

Coprocessor Unusable, Reserved Instruction

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.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Branch on Equal BEQ

31 26	25 21	20 16	15 0
BEQ 000100	rs	rt	offset
6	5	5	16

Format: `BEQ rs, rt, offset` **MIPS32**

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:

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

Operation:

```
I:      target_offset ← sign_extend(offset || $0^2$)
           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.

BEQ r0, r0 offset, expressed as B offset, is the assembly idiom used to denote an unconditional branch.

Branch on Equal Likely **BEQL**

31 26	25 21	20 16	15 0
BEQL 010100	rs	rt	offset
6	5	5	16

Format: `BEQL rs, rt, offset` **MIPS32**

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:

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

Operation:

```
I:     target_offset ← sign_extend(offset || $0^2$)
          condition ← (GPR[rs] = GPR[rt])
I+1:   if condition then
          PC ← PC + target_offset
       else
          NullifyCurrentInstruction()
       endif
```

Exceptions:

None

Branch on Equal Likely (cont.) BEQL

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.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS 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` **MIPS32**

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:

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

Operation:

```
I:     target_offset ← sign_extend(offset || $0^2$)
       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 000001	rs	BGEZAL 10001	offset
6	5	5	16

Format: `BGEZAL rs, offset` **MIPS32**

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 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:

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

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 **UNPREDICTABLE**. 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 || $0^{2}$)
        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.

BGEZAL r0, offset, expressed as BAL offset, is the assembly idiom used to denote a PC-relative branch and link. BAL is used in a manner similar to JAL, but provides PC-relative addressing and a more limited target PC range.

Branch on Greater Than or Equal to Zero and Link Likely — BGEZALL

31 — 26	25 — 21	20 — 16	15 — 0
REGIMM 000001	rs	BGEZALL 10011	offset
6	5	5	16

Format: `BGEZALL rs, offset` **MIPS32**

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 **UNPREDICTABLE**. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

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

Operation:

```
I:      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
        else
           NullifyCurrentInstruction()
        endif
```

Exceptions:

None

Branch on Greater Than or Equal to Zero and Link Likely (con’t.) BGEZALL

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.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS 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` **MIPS32**

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:

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

Operation:

```
I:      target_offset ← sign_extend(offset || $0^2$)
        condition ← GPR[rs] ≥ $0^{GPRLEN}$
I+1:    if condition then
           PC ← PC + target_offset
        else
           NullifyCurrentInstruction()
        endif
```

Exceptions:

None

Branch on Greater Than or Equal to Zero Likely (cont.) BGEZL

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.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Branch on Greater Than Zero **BGTZ**

31 26	25 21	20 16	15 0
BGTZ 000111	rs	0 00000	offset
6	5	5	16

Format: `BGTZ rs, offset` **MIPS32**

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:

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

Operation:

```
I:     target_offset ← sign_extend(offset || $0^2$)
       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 010111	rs	0 00000	offset
6	5	5	16

Format: `BGTZL rs, offset` **MIPS32**

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:

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

Operation:

```
I:     target_offset ← sign_extend(offset || $0^2$)
       condition ← GPR[rs] > $0^{GPRLEN}$
I+1:   if condition then
          PC ← PC + target_offset
       else
          NullifyCurrentInstruction()
       endif
```

Exceptions:

None

Branch on Greater Than Zero Likely (cont.) BGTZL

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.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Branch on Less Than or Equal to Zero BLEZ

31 26	25 21	20 16	15 0
BLEZ 000110	rs	0 00000	offset
6	5	5	16

Format: `BLEZ rs, offset` **MIPS32**

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:

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

Operation:

```
I:     target_offset ← sign_extend(offset || $0^2$)
       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 010110	rs	0 00000	offset
6	5	5	16

Format: `BLEZL rs, offset` **MIPS32**

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:

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

Operation:

```
I:      target_offset ← sign_extend(offset || $0^2$)
        condition ← GPR[rs] ≤ $0^{GPRLEN}$
I+1:    if condition then
           PC ← PC + target_offset
        else
           NullifyCurrentInstruction()
        endif
```

Exceptions:

None

Branch on Less Than or Equal to Zero Likely (cont.) BLEZL

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.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Branch on Less Than Zero BLTZ

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZ 00000	offset
6	5	5	16

Format: `BLTZ rs, offset` **MIPS32**

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:

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

Operation:

```
I:      target_offset ← sign_extend(offset || $0^2$)
        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` **MIPS32**

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 UNPREDICTABLE. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

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

Operation:

```
I:      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 000001	rs	BLTZALL 10010	offset
6	5	5	16

Format: `BLTZALL rs, offset` **MIPS32**

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 UNPREDICTABLE. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

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

Operation:

```
I:      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
        else
           NullifyCurrentInstruction()
        endif
```

Exceptions:

None

Branch on Less Than Zero and Link Likely (cont.) BLTZALL

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.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS 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` **MIPS32**

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:

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

Operation:

```
I:      target_offset ← sign_extend(offset || $0^2$)
        condition ← GPR[rs] < $0^{GPRLEN}$
I+1:    if condition then
           PC ← PC + target_offset
        else
           NullifyCurrentInstruction()
        endif
```

Exceptions:

None

Branch on Less Than Zero Likely (cont.) BLTZL

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.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Branch on Not Equal **BNE**

31 26	25 21	20 16	15 0
BNE 000101	rs	rt	offset
6	5	5	16

Format: `BNE rs, rt, offset` **MIPS32**

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:

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

Operation:

```
I:     target_offset ← sign_extend(offset || $0^2$)
       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 010101	rs	rt	offset
6	5	5	16

Format: `BNEL rs, rt, offset` **MIPS32**

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:

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

Operation:

```
I:      target_offset ← sign_extend(offset || $0^2$)
        condition ← (GPR[rs] ≠ GPR[rt])
I+1:    if condition then
           PC ← PC + target_offset
        else
           NullifyCurrentInstruction()
        endif
```

Exceptions:

None

Branch on Not Equal Likely (cont.) BNEL

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.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Breakpoint BREAK

31 26	25 6	5 0
SPECIAL 000000	code	BREAK 001101
6	20	6

Format: `BREAK` **MIPS32**

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

Floating Point Compare C.cond.fmt

31 26	25 21	20 16	15 11	10 8	7	6	5 4	3 0
COP1 010001	fmt	ft	fs	cc	0	A 0	FC 11	cond
6	5	5	5	3	1	1	2	4

Format: `C.cond.S fs, ft (cc = 0 implied)` **MIPS32**
`C.cond.D fs, ft (cc = 0 implied)` **MIPS32**
`C.cond.S cc, fs, ft` **MIPS32**
`C.cond.D cc, fs, ft` **MIPS32**

Purpose:

To compare FP values and record the Boolean result in a condition code

Description: `cc` ← `fs` *compare_cond* `ft`

The value in FPR *fs* is compared to the value in FPR *ft*; the values are in format *fmt*. The comparison is exact and neither overflows nor underflows.

If the comparison specified by $cond_{2..1}$ is true for the operand values, the result is true; otherwise, the result is false. If no exception is taken, the result is written into condition code *CC*; true is 1 and false is 0.

If one of the values is an SNaN, or $cond_3$ is set and at least one of the values is a QNaN, an Invalid Operation condition is raised and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written and an Invalid Operation exception is taken immediately. Otherwise, the Boolean result is written into condition code *CC*.

There are four mutually exclusive ordering relations for comparing floating point values; one relation is always true and the others are false. The familiar relations are *greater than*, *less than*, and *equal*. In addition, the IEEE floating point standard defines the relation *unordered*, which is true when at least one operand value is NaN; NaN compares unordered with everything, including itself. Comparisons ignore the sign of zero, so +0 equals -0.

The comparison condition is a logical predicate, or equation, of the ordering relations such as *less than or equal*, *equal*, *not less than*, or *unordered or equal*. Compare distinguishes among the 16 comparison predicates. The Boolean result of the instruction is obtained by substituting the Boolean value of each ordering relation for the two FP values in the equation. If the *equal* relation is true, for example, then all four example predicates above yield a true result. If the *unordered* relation is true then only the final predicate, *unordered or equal*, yields a true result.

Logical negation of a compare result allows eight distinct comparisons to test for the 16 predicates as shown in . Each mnemonic tests for both a predicate and its logical negation. For each mnemonic, *compare* tests the truth of the first predicate. When the first predicate is true, the result is true as shown in the “If Predicate Is True” column, and the second predicate must be false, and vice versa. (Note that the False predicate is never true and False/True do not follow the normal pattern.)

The truth of the second predicate is the logical negation of the instruction result. After a compare instruction, test for the truth of the first predicate can be made with the Branch on FP True (BC1T) instruction and the truth of the second can be made with Branch on FP False (BC1F).

Floating Point Compare (cont.) C.cond.fmt

Table 12-22 shows another set of eight compare operations, distinguished by a $cond_3$ value of 1 and testing the same 16 conditions. For these additional comparisons, if at least one of the operands is a NaN, including Quiet NaN, then an Invalid Operation condition is raised. If the Invalid Operation condition is enabled in the *FCSR*, an Invalid Operation exception occurs.

Table 12-22 FPU Comparisons Without Special Operand Exceptions

Instruction	**Comparison Predicate**					**Comparison CC Result**		**Instruction**	
Cond Mnemonic	**Name of Predicate and Logically Negated Predicate (Abbreviation)**	**Relation Values**				**If Predicate Is True**	**Inv Op Excp. if QNaN ?**	**Condition Field**	
		>	<	=	**?**			**3**	**2..0**
F	False [this predicate is always False]	F	F	F	F	F	No	0	0
	True (T)	T	T	T	T				
UN	Unordered	F	F	F	T	T			1
	Ordered (OR)	T	T	T	F	F			
EQ	Equal	F	F	T	F	T			2
	Not Equal (NEQ)	T	T	F	T	F			
UEQ	Unordered or Equal	F	F	T	T	T			3
	Ordered or Greater Than or Less Than (OGL)	T	T	F	F	F			
OLT	Ordered or Less Than	F	T	F	F	T			4
	Unordered or Greater Than or Equal (UGE)	T	F	T	T	F			
ULT	Unordered or Less Than	F	T	F	T	T			5
	Ordered or Greater Than or Equal (OGE)	T	F	T	F	F			
OLE	Ordered or Less Than or Equal	F	T	T	F	T			6
	Unordered or Greater Than (UGT)	T	F	F	T	F			
ULE	Unordered or Less Than or Equal	F	T	T	T	T			7
	Ordered or Greater Than (OGT)	T	F	F	F	F			
Key: ? = *unordered*, > = *greater than*, < = *less than*, = is *equal*, T = *True*, F = False									

Floating Point Compare (cont.) C.cond.fmt

Table 12-23 FPU Comparisons With Special Operand Exceptions for QNaNs

Instruction	Comparison Predicate					Comparison CC Result		Instructio n	
Cond Mnemonic	Name of Predicate and Logically Negated Predicate (Abbreviation)	Relation Values				If Predicate Is True	Inv Op Excp If QNaN?	Condition Field	
		>	<	=	?			3	2..0
SF	Signaling False [this predicate always False]	F	F	F	F	F	Yes	1	0
	Signaling True (ST)	T	T	T	T				
NGLE	Not Greater Than or Less Than or Equal	F	F	F	T	T			1
	Greater Than or Less Than or Equal (GLE)	T	T	T	F	F			
SEQ	Signaling Equal	F	F	T	F	T			2
	Signaling Not Equal (SNE)	T	T	F	T	F			
NGL	Not Greater Than or Less Than	F	F	T	T	T			3
	Greater Than or Less Than (GL)	T	T	F	F	F			
LT	Less Than	F	T	F	F	T			4
	Not Less Than (NLT)	T	F	T	T	F			
NGE	Not Greater Than or Equal	F	T	F	T	T			5
	Greater Than or Equal (GE)	T	F	T	F	F			
LE	Less Than or Equal	F	T	T	F	T			6
	Not Less Than or Equal (NLE)	T	F	F	T	F			
NGT	Not Greater Than	F	T	T	T	T			7
	Greater Than (GT)	T	F	F	F	F			
Key: ? = *unordered,* > = *greater than,* < = *less than,* = is *equal,* T = *True,* F = *False*									

Floating Point Compare (cont.) C.cond.fmt

Restrictions:

The fields *fs* and *ft* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

Operation:

```
if SNaN(ValueFPR(fs, fmt)) or SNaN(ValueFPR(ft, fmt)) or
   QNaN(ValueFPR(fs, fmt)) or QNaN(ValueFPR(ft, fmt)) then
   less ← false
   equal ← false
   unordered ← true
   if (SNaN(ValueFPR(fs,fmt)) or SNaN(ValueFPR(ft,fmt))) or
   (cond$_3$ and (QNaN(ValueFPR(fs,fmt)) or QNaN(ValueFPR(ft,fmt)))) then
      SignalException(InvalidOperation)
   endif
else
   less ← ValueFPR(fs, fmt) $<_{fmt}$ ValueFPR(ft, fmt)
   equal ← ValueFPR(fs, fmt) $=_{fmt}$ ValueFPR(ft, fmt)
   unordered ← false
endif
condition ← (cond$_2$ and less) or (cond$_1$ and equal)
      or (cond$_0$ and unordered)
SetFPConditionCode(cc, condition)
```

Floating Point Compare (cont.) C.cond.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

Programming Notes:

FP computational instructions, including compare, that receive an operand value of Signaling NaN raise the Invalid Operation condition. Comparisons that raise the Invalid Operation condition for Quiet NaNs in addition to SNaNs permit a simpler programming model if NaNs are errors. Using these compares, programs do not need explicit code to check for QNaNs causing the *unordered* relation. Instead, they take an exception and allow the exception handling system to deal with the error when it occurs. For example, consider a comparison in which we want to know if two numbers are equal, but for which *unordered* would be an error.

```
# comparisons using explicit tests for QNaN
    c.eq.d $f2,$f4# check for equal
    nop
    bc1t   L2     # it is equal
    c.un.d $f2,$f4# it is not equal,
                  # but might be unordered
    bc1t   ERROR  # unordered goes off to an error handler
# not-equal-case code here
    ...
# equal-case code here
L2:
# --------------------------------------------------------------
# comparison using comparisons that signal QNaN
    c.seq.d $f2,$f4  # check for equal
    nop
    bc1t   L2        # it is equal
    nop
# it is not unordered here
    ...
# not-equal-case code here
    ...
# equal-case code here
```

Perform Cache Operation CACHE

31 26	25 21	20 16	15 0
CACHE 101111	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 the following ways based on the operation to be performed and the type of cache as described in the following table.

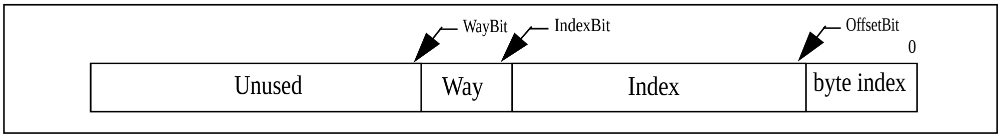
Table 12-24 Usage of Effective Address

Operation Requires an	Type of Cache	Usage of Effective Address
Address	Physical	The effective address is used to address the cache. An address translation is performed on the effective address (with the possibility that a TLB Refill or TLB Invalid exception might occur)

Index	N/A	The effective 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 ← $\mathrm{Addr}_{\mathrm{WayBit-1..IndexBit}}$
		Index ← $\mathrm{Addr}_{\mathrm{IndexBit-1..OffsetBit}}$
		Indexed cache instructions referring to disabled or non-existent ways are ignored.

Perform Cache Operation **CACHE**

Figure 12-4 Usage of Address Fields to Select Index and Way



A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur on any operation. For index operations (where the address is used to index the cache but need not match the cache tag) software should use unmapped addresses 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.

A Cache Error exception may occur as a byproduct of some operations performed by this instruction. For example, if a Writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error.

An address Error Exception (with cause code equal AdEL) occurs if the effective address references a portion of the kernel address space which would normally result in such an exception.

Data watch is not triggered by a cache instruction whose address matches the Watch register address match conditions.

Bits [17:16] of the instruction specify the cache on which to perform the operation, as follows:

Table 12-25 Encoding of Bits[17:16] of CACHE Instruction

Code	Name	Cache
2#00	I	Primary Instruction
2#01	D	Primary Data
2#10	T	Not supported
2#11	S	Not supported

Bits [20:18] of the instruction specify the operation to perform. On Index Load Tag and Index Store Data operations, the specific double-word that is addressed is loaded into / read from the DataLo and DataHi registers. All other cache instructions are line-based and the word and byte indexes will not affect their operation.

Table 12-26 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	Implemented?
2#000	I	Index Invalidate	Index	Set the state of the cache block at the specified index to invalid. This encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.	Yes
	D	Index Writeback Invalidate	Index	If the state of the cache block at the specified index is valid and dirty, write the block back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. This encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. Note that Index Store Tag should be used to initialize the cache at powerup.	Yes
	S, T	Reserved	Index		No
2#001	I,D	Index Load Tag	Index	Read the tag for the cache block at the specified way and index into the *TagLo* and *TagHi* Coprocessor 0 register. Also read the data corresponding to the byte index into the *DataLo* and *DataHi* registers. When loading data into the *DataLo* and *DataHi* registers the lower three bits of the byte index are ignored in order to obtain an aligned double word access to the cache.	Yes
2#010	I,D	Index Store Tag	Index	Write the tag for the cache block at the specified index from the *TagLo* and *TagHi* Coprocessor 0 registers. This encoding may be used by software to initialize the entire instruction or data caches by stepping through all valid indices. Doing so requires that the *TagLo* and *TagHi* registers associated with the cache be initialized first.	Yes

Table 12-26 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	Implemented?
2#011	All	Reserved	Unspecified	Executed as a no-op.	No
2#100	I, D	Hit Invalidate	Address	If the cache block contains the specified address, set the state of the cache block to invalid. This encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache.	Yes
	S, T	Reserved	Address		No
2#101	I	Fill	Address	Fill the cache from the specified address. The cache line is fecthed only if it is not already in the cache.	Yes
	D	Hit Writeback Invalidate	Address	If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. This encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache.	Yes
	S, T	Reserved	Address		No
2#110	D	Hit Writeback	Address	If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state.	Yes
	S, T	Reserved	Address		No

Table 12-26 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	Implemented?
2#111	I, D	Fetch and Lock	Address	If the cache does not contain the specified address, fill it from memory, performing a writeback if required, and set the state to valid and locked. If the cache already contains the specified address, set the state to locked. The way selected on fill from memory is the least recently used. The lock state is cleared by executing an Index Invalidate, Index Writeback Invalidate, Hit Invalidate, or Hit Writeback Invalidate operation to the locked line, or via an Index Store Tag operation with the lock bit reset in the *TagLo* register.	Yes

Table 12-27 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST] Set

Code	Caches	Name	Effective Address Operand Type	Operation	Implemented?
2#001	I, D	Index Load WS	Index	Read the WS RAM at the specified index into the *TagLo* Coprocessor 0 register.	Yes
2#010	I, D	Index Store WS	Index	Update the WS RAM at the specified index from the *TagLo* Coprocessor 0 register.	Yes
2#011	I, D	Index Store Data	Index	Write the *DataHi* and *DataLo* Coprocessor 0 register contents at the way and byte index specified. The lower three bits of the byte index are ignored in order to obtain an aligned double word access to the cache.	Yes
All Others	All	Reserved	Unspecified	Executed as no-op.	No

Perform Cache Operation (cont.) CACHE

Restrictions:

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

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

Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, uncached) ← AddressTranslation(vAddr, DataReadReference)
CacheOp(op, vAddr, pAddr)
```

Exceptions:

TLB Refill Exception.

TLB Invalid Exception

Coprocessor Unusable Exception

Address Error Exception

Cache Error Exception

Bus Error Exception

Fixed Point Ceiling Convert to Long Fixed Point CEIL.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CEIL.L 001010
6	5	5	5	5	6

Format: `CEIL.L.S   fd, fs` **MIPS64**
`CEIL.L.D   fd, fs` **MIPS64**

Purpose:

To convert an FP value to 64-bit fixed point, rounding up

Description: `fd ← convert_and_round(fs)`

The value in FPR *fs*, in format *fmt*, is converted to a value in 64-bit long fixed point format and rounding toward +∞ (rounding mode 2). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, a d the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}–1$, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Fixed Point Ceiling Convert to Long Fixed Point (cont.) CEIL.L.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow

Floating Point Ceiling Convert to Word Fixed Point CEIL.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CEIL.W 001110
6	5	5	5	5	6

Format: `CEIL.W.S fd, fs` **MIPS32**
`CEIL.W.D fd, fs` **MIPS32**

Purpose:

To convert an FP value to 32-bit fixed point, rounding up

Description: `fd ← convert_and_round(fs)`

The value in FPR *fs,* in format *fmt,* is converted to a value in 32-bit word fixed point format and rounding toward +∞ (rounding mode 2). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to 2^{31}-1, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, 2^{31}–1, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow

Move Control Word From Floating Point CFC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	CF 00010	rt	fs	0 000 0000 0000
6	5	5	5	11

Format: `CFC1 rt, fs` **MIPS32**

Purpose:

To copy a word from an FPU control register to a GPR

Description: `rt ← FP_Control[fs]`

Copy the 32-bit word from FP (coprocessor 1) control register *fs* into GPR *rt*, sign-extending it to 64 bits.

Restrictions:

There are a few control registers defined for the floating point unit. The result is **UNPREDICTABLE** if *fs* specifies a register that does not exist.

Operation:

```
if fs = 0 then
   temp ← FIR
elseif fs = 25 then
   temp ← $0^{24}$ || $FCSR_{31..25}$ || $FCSR_{23}$
elseif fs = 26 then
   temp ← $0^{14}$ || $FCSR_{17..12}$ || $0^{5}$ || $FCSR_{6..2}$ || $0^{2}$
elseif fs = 28 then
   temp ← $0^{20}$ || $FCSR_{11.7}$ || $0^{4}$ || $FCSR_{24}$ || $FCSR_{1..0}$
elseif fs = 31 then
   temp ← FCSR
else
   temp ← UNPREDICTABLE
endif
GPR[rt] ← sign_extend(temp)
```

Move Control Word From Floating Point (cont.) CFC1

Exceptions:

Coprocessor Unusable, Reserved Instruction

Move Control Word From Coprocessor 2 CFC2

31 26	25 21	20 16	15 11	10 0
COP2 010010	CF 00010	rt	rd	0 000 0000 0000
6	5	5	5	11

Format: `CFC2 rt, rd` **MIPS32**

Purpose:

To copy a word from a Coprocessor 2 control register to a GPR

Description: `rt ← CCR[2,rd]`

Copy the 32-bit word from Coprocessor 2 control register *rd* into GPR *rt,* sign-extending it to 64 bits.

Restrictions:

The result is **UNPREDICTABLE** if *fs* specifies a register that does not exist.

Operation:

```
temp ← CCR[2,rd]
GPR[rt] ← sign_extend(temp)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Count Leading Ones in Word CLO

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

Format: `CLO rd, rs` `MIPS32`

Purpose:

To Count the number of leading ones in a word

Description: `rd ← count_leading_ones rs`

Bits 31..0 of GPR *rs* are 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 of bits 31..0 were set in GPR *rs*, the result written to GPR *rd* is 32.

Restrictions:

To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the *rt* and *rd* fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values.

If GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[rs]) then
   UNPREDICTABLE
endif
temp ← 32
for i in 31 .. 0
   if $GPR[rs]_i$ = 0 then
      temp ← 31 - i
      break
   endif
endfor
GPR[rd] ← temp
```

Exceptions:

None

Count Leading Zeros in Word **CLZ**

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

Format: `CLZ rd, rs` `MIPS32`

Purpose

Count the number of leading zeros in a word

Description: `rd ← count_leading_zeros rs`

Bits 31..0 of GPR *rs* are 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 *rt* is 32.

Restrictions:

To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the *rt* and *rd* fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values.

If GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[rs]) then
   UNPREDICTABLE
endif
temp ← 32
for i in 31 .. 0
   if $GPR[rs]_i$ = 1 then
      temp ← 31 - i
      break
   endif
endfor
GPR[rd] ← temp
```

Exceptions:

None

Move Control Word to Floating Point CTC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	CT 00110	rt	fs	0 000 0000 0000
6	5	5	5	11

Format: `CTC1 rt, fs` **MIPS32**

Purpose:

To copy a word from a GPR to an FPU control register

Description: `FP_Control[fs] ← rt`

Copy the low word from GPR *rt* into the FP (coprocessor 1) control register indicated by *fs*.

Writing to the floating point *Control/Status* register, the *FCSR*, causes the appropriate exception if any *Cause* bit and its corresponding *Enable* bit are both set. The register is written before the exception occurs. Writing to *FEXR* to set a cause bit whose enable bit is already set, or writing to *FENR* to set an enable bit whose cause bit is already set causes the appropriate exception. The register is written before the exception occurs.

Restrictions:

There are a few control registers defined for the floating point unit. The result is **UNPREDICTABLE** if *fs* specifies a register that does not exist.

Move Control Word to Floating Point (cont.) CTC1

Operation:

```
temp ← $GPR[rt]_{31..0}$
if fs = 25 then
    if $temp_{31..8} \neq 0^{24}$ then
        **UNPREDICTABLE**
    else
        FCSR ← $temp_{7..1}$ || $FCSR_{24}$ || $temp_0$ || $FCSR_{22..0}$
    endif
elseif fs = 26 then
    if $temp_{22..18} \neq 0$ then
        **UNPREDICTABLE**
    else
        FCSR ← $FCSR_{31..18}$ || $temp_{17..12}$ || $FCSR_{11..7}$ ||
        $temp_{6..2}$ || $FCSR_{1..0}$
    endif
elseif fs = 28 then
    if $temp_{22..18} \neq 0$ then
        **UNPREDICTABLE**
    else
        FCSR ← $FCSR_{31..25}$ || $temp_2$ || $FCSR_{23..12}$ || $temp_{11..7}$
        || $FCSR_{6..2}$ || $temp_{1..0}$
    endif
elseif fs = 31 then
    if $temp_{22..18} \neq 0$ then
        **UNPREDICTABLE**
    else
        FCSR ← temp
    endif
else
    **UNPREDICTABLE**
endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Division-by-zero, Inexact, Overflow, Underflow

Move Control Word to Coprocessor 2 CTC2

31 26	25 21	20 16	15 11	10 0
COP2 010010	CT 00110	rt	rd	0 000 0000 0000
6	5	5	5	11

Format: `CTC2 rt, rd` **MIPS32**

Purpose:

To copy a word from a GPR to a Coprocessor 2 control register

Description: `CCR[2,rd] ← rt`

Copy the low word from GPR *rt* into the Coprocessor 2 control register indicated by *rd*.

Restrictions:

The result is **UNPREDICTABLE** if *rd* specifies a register that does not exist.

Operation:

```
temp ← GPR[rt]$_{31..0}$
CCR[2,rd] ← temp
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Convert to Double Floating Point **CVT.D.fmt**

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.D 100001
6	5	5	5	5	6

Format: `CVT.D.S fd, fs` **MIPS32**
`CVT.D.W fd, fs` **MIPS32**
`CVT.D.L fd, fs` **MIPS64**

Purpose:

To convert an FP or fixed point value to double FP

Description: `fd ← convert_and_round(fs)`

The value in FPR *fs*, in format *fmt*, is converted to a value in double floating point format and rounded according to the current rounding mode in *FCSR*. The result is placed in FPR *fd*. If *fmt* is S or W, then the operation is always exact.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *fmt* and *fd* for double floating point—if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR (fd, D, ConvertFmt(ValueFPR(fs, fmt), fmt, D))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact

Floating Point Convert to Long Fixed Point CVT.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.L 100101
6	5	5	5	5	6

Format: `CVT.L.S fd, fs` **MIPS64**
`CVT.L.D fd, fs` **MIPS64**

Purpose:

To convert an FP value to a 64-bit fixed point

Description: `fd ← convert_and_round(fs)`

Convert the value in format *fmt* in FPR *fs* to long fixed point format and round according to the current rounding mode in *FCSR*. The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}–1$, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *fmt* and *fd* for long fixed point—if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR (fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow

Floating Point Convert to Single Floating Point **CVT.S.fmt**

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.S 100000
6	5	5	5	5	6

Format: `CVT.S.D fd, fs` **MIPS32**
`CVT.S.W fd, fs` **MIPS32**
`CVT.S.L fd, fs` **MIPS64**

Purpose:

To convert an FP or fixed point value to single FP

Description: `fd ← convert_and_round(fs)`

The value in FPR *fs*, in format *fmt*, is converted to a value in single floating point format and rounded according to the current rounding mode in *FCSR*. The result is placed in FPR *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *fmt* and *fd* for single floating point. If they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR(fd, S, ConvertFmt(ValueFPR(fs, fmt), fmt, S))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow, Underflow

Floating Point Convert to Word Fixed Point — CVT.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.W 100100
6	5	5	5	5	6

Format:

```
CVT.W.S fd, fs                                  MIPS32
CVT.W.D fd, fs                                  MIPS32
```

Purpose:

To convert an FP value to 32-bit fixed point

Description: `fd ← convert_and_round(fs)`

The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format and rounded according to the current rounding mode in *FCSR*. The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}–1$, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *fmt* and *fd* for word fixed point—if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow

Doubleword Add **DADD**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DADD 101100
6	5	5	5	5	6

Format: `DADD rd, rs, rt` **MIPS64**

Purpose:

To add 64-bit integers. If overflow occurs, then trap.

Description: `rd ← rs + rt`

The 64-bit doubleword value in GPR *rt* is added to the 64-bit value in GPR *rs* to produce a 64-bit result. If the addition results in 64-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 64-bit result is placed into GPR *rd*.

Restrictions:

Operation:

```
temp ← (GPR[rs]$_{63}$||GPR[rs]) + (GPR[rt]$_{63}$||GPR[rt])
if (temp$_{64}$ ≠ temp$_{63}$) then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp$_{63..0}$
endif
```

Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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

Doubleword Add Immediate DADDI

31 26	25 21	20 16	15 0
DADDI 011000	rs	rt	immediate
6	5	5	16

Format: `DADDI rt, rs, immediate` **MIPS64**

Purpose:

To add a constant to a 64-bit integer. If overflow occurs, then trap.

Description: `rt ← rs + immediate`

The 16-bit signed *immediate* is added to the 64-bit value in GPR *rs* to produce a 64-bit result. If the addition results in 64-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 64-bit result is placed into GPR *rt*.

Restrictions:

Operation:

```
temp ← ($GPR[rs]_{63}$||GPR[rs]) + sign_extend(immediate)
if ($temp_{64} \neq temp_{63}$) then
   SignalException(IntegerOverflow)
else
   GPR[rt] ← $temp_{63..0}$
endif
```

Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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

Doubleword Add Immediate Unsigned **DADDIU**

31 26	25 21	20 16	15 0
DADDIU 011001	rs	rt	immediate
6	5	5	16

Format: `DADDIU rt, rs, immediate` **MIPS64**

Purpose:

To add a constant to a 64-bit integer

Description: `rt ← rs + immediate`

The 16-bit signed *immediate* is added to the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

Operation:

```
GPR[rt] ← GPR[rs] + sign_extend(immediate)
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Doubleword Add Unsigned DADDU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DADDU 101101
6	5	5	5	5	6

Format: `DADDU   rd, rs, rt` **MIPS64**

Purpose:

To add 64-bit integers

Description: `rd ← rs + rt`

The 64-bit doubleword value in GPR *rt* is added to the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

Operation:

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

Exceptions:

Reserved Instruction

Programming Notes:

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

Count Leading Ones in Doubleword DCLO

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	rd	0 00000	DCLO 100101
6	5	5	5	5	6

Format: `DCLO rd, rs` **MIPS64**

Purpose:

To count the number of leading ones in a doubleword

Description: `rd ← count_leading_ones rs`

The 64-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 64 bits were set in GPR *rs*, the result written to GPR *rd* is 64.

Restrictions:

To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the *rt* and *rd* fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values.

Operation:

```
temp <- 64
for i in 63.. 0
    if $GPR[rs]_i$ = 1 then
        temp <- 63 - i
        break
    endif
endfor
GPR[rd] <- temp
```

Exceptions:

None

Count Leading Zeros in Doubleword DCLZ

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	rd	0 00000	DCLZ 100100
6	5	5	5	5	6

Format: `DCLZ rd, rs` **MIPS64**

Purpose:

To count the number of leading zeros in a doubleword

Description: `rd ← count_leading_zeros rs`

The 64-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 64.

Restrictions:

To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the *rt* and *rd* fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the *rt* and *rd* fields of the instruction contain different values.

Operation:

```
temp <- 64
for i in 63.. 0
   if GPR[rs]$_i$ = 0 then
      temp <- 63 - i
      break
   endif
endfor
GPR[rd] <- temp
```

Exceptions:

None

Doubleword Divide DDIV

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DDIV 011110
6	5	5	10	6

Format: `DDIV rs, rt` **MIPS64**

Purpose:

To divide 64-bit signed integers

Description: `(LO, HI) ← rs / rt`

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

No arithmetic exception occurs under any circumstances.

Restrictions:

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

Operation:

```
LO ← GPR[rs] div GPR[rt]
HI ← GPR[rs] mod GPR[rt]
```

Exceptions:

Reserved Instruction

Programming Notes:

See “Programming Notes” for the DIV instruction.

Doubleword Divide Unsigned DDIVU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DDIVU 011111
6	5	5	10	6

Format: `DDIVU rs, rt` **MIPS64**

Purpose:

To divide 64-bit unsigned integers

Description: `(LO, HI) ← rs / rt`

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

No arithmetic exception occurs under any circumstances.

Restrictions:

If the divisor in GPR *rt* is zero, the arithmetic result value is undefined.

Operation:

```
q ← (0 || GPR[rs]) div (0 || GPR[rt])
r ← (0 || GPR[rs]) mod (0 || GPR[rt])
LO ← $q_{63..0}$
HI ← $r_{63..0}$
```

Exceptions:

Reserved Instruction

Programming Notes:

See “Programming Notes” for the DIV instruction.

Debug Exception Return DERET

31 26	25	24 6	5 0
COP0	CO	0	DERET
010000	1	000 0000 0000 0000 0000	011111
6	1	19	6

Format: `DERET` `EJTAG`

Purpose:

To Return from a debug exception.

Description:

DERET returns from Debug Mode and resumes non-debug execution at the instruction whose address is contained in the *DEPC* register. DERET does not execute the next instruction (i.e. it has no delay slot).

Restrictions:

A DERET placed between an LL and SC instruction does not cause the SC to fail.

If the DEPC register with the return address for the DERET was modified by an MTC0 or a DMTC0 instruction, a CP0 hazard hazard exists that must be removed via software insertion of the apporpriate number of SSNOP instructions.

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.

This instruction is legal only if the processor is executing in Debug Mode.The operation of the processor is **UNDEFINED** if a DERET is executed in the delay slot of a branch or jump instruction.

Debug Exception Return (cont.) DERET

Operation:

```
Debug_DM ← 0
Debug_IEXI ← 0
if IsMIPS16Implemented() then
   PC ← DEPC_63..1 ‖ 0
   ISAMode ← 0 ‖ DEPC_0
else
   PC ← DEPC
endif
```

Exceptions:

Coprocessor Unusable Exception
Reserved Instruction Exception

Divide Word **DIV**

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

Format: `DIV rs, rt` **MIPS32**

Purpose:

To divide a 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 sign-extended and placed into special register *LO* and the 32-bit remainder is sign-extended and placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

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

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

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
    UNPREDICTABLE
endif
q  ← GPR[rs]$_{31..0}$ div GPR[rt]$_{31..0}$
LO ← sign_extend(q$_{31..0}$)
r  ← GPR[rs]$_{31..0}$ mod GPR[rt]$_{31..0}$
HI ← sign_extend(r$_{31..0}$)
```

Exceptions:

None

Divide Word (cont.) DIV

Programming Notes:

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

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

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

Floating Point Divide DIV.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	DIV 000011
6	5	5	5	5	6

Format: `DIV.S fd, fs, ft` **MIPS32**
`DIV.D fd, fs, ft` **MIPS32**

Purpose:

To divide FP values

Description: `fd ← fs / ft`

The value in FPR *fs* is divided by the value in FPR *ft*. The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR (fd, fmt, ValueFPR(fs, fmt) / ValueFPR(ft, fmt))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Invalid Operation, Unimplemented Operation, Division-by-zero, Overflow, Underflow

Divide Unsigned Word **DIVU**

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DIVU 011011
6	5	5	10	6

Format: `DIVU rs, rt` **MIPS32**

Purpose:

To divide a 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 sign-extended and placed into special register *LO* and the 32-bit remainder is sign-extended and placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

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

If the divisor in GPR *rt* is zero, the arithmetic result value is undefined.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
   UndefinedResult()
endif
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(q$_{31..0}$)
HI ← sign_extend(r$_{31..0}$)
```

Exceptions:

None

Programming Notes:

See “Programming Notes” for the DIV instruction.

Doubleword Move from Coprocessor 0 DMFC0

31 26	25 21	20 16	15 11	10 3	2 0
COP0 010000	DMF 00001	rt	rd	0 0000 0000	sel
6	5	5	5	8	3

Format: `DMFC0 rt, rd` **MIPS64**
`DMFC0 rt, rd, sel` **MIPS64**

Purpose:

To move the contents of a coprocessor 0 register to a general purpose register (GPR).

Description: `rt ← CPR[0,rd,sel]`

The contents of the coprocessor 0 register are loaded into GPR *rt*. Note that not all coprocessor 0 registers support the *sel* field. In those instances, the *sel* field must be zero.

Restrictions:

The results are **UNPREDICTABLE** if coprocessor 0 does not contain a register as specified by *rd* and *sel*, or if the coprocessor 0 register specified by *rd* and *sel* is a 32-bit register.

Operation:

```
datadoubleword ← CPR[0,rd,sel]
GPR[rt] ← datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Doubleword Move from Floating Point DMFC1

31 26	25 21	20 16	15 11	10 0
COP1	DMF	rt	fs	0
010001	00001			000 0000 0000
6	5	5	5	11

Format: `DMFC1 rt,fs` **MIPS64**

Purpose:

To move a doubleword from an FPR to a GPR.

Description: `rt← fs`

The contents of FPR *fs* are loaded into GPR *rt*.

Restrictions:

None

Operation:

```
datadoubleword ←ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)
GPR[rt] ←datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Doubleword Move from Coprocessor 2 DMFC2

31 26	25 21	20 16	15 11	10 3	2 0
COP2 010010	DMF 00001	rt	rd	0 000 0000 0	sel
6	5	5	5	8	3

Format: `DMFC2 rt, rd` **MIPS64**
`DMFC2, rt, rd,sel` **MIPS64**

Purpose:

To move a doubleword from a coprocessor 2 register to a GPR.

Description: `rt ← CPR[2, rd, sel]`

The contents of the coprocessor 2 register specified by the *rd* and *sel* fields are loaded into GPR *rt*. Note that not all coprocessor 2 registers may support the *sel* field. In those instances, the *sel* field must be zero.

Restrictions:

The results are **UNPREDICTABLE** if coprocessor 2 does not contain a register as specified by *rd* and *sel*, or if the coprocessor 2 register specified by *rd* and *sel* is a 32-bit register.

Operation:

```
datadoubleword ←CPR[2,rd,sel]
GPR[rt] ←datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Doubleword Move to Coprocessor 0 **DMTC0**

31 26	25 21	20 16	15 11	10 3	2 0
COP0 010000	DMT 00101	rt	rd	0 0000 0000	sel
6	5	5	5	8	3

Format: `DMTC0 rt, rd` **MIPS64**
`DMTC0 rt, rd, sel` **MIPS64**

Purpose:

To move a doubleword from a GPR to a coprocessor 0 register.

Description: `CPR[0,rd,sel] ← rt`

The contents of GPR *rt* are loaded into the coprocessor 0 register specified in the *rd* and *sel* fields. Note that not all coprocessor 0 registers support the *sel* field. In those instances, the *sel* field must be zero.

Restrictions:

The results are **UNPREDICTABLE** if coprocessor 0 does not contain a register as specified by *rd* and *sel*, or if the coprocessor 0 register specified by *rd* and *sel* is a 32-bit register.

Operation:

```
datadoubleword ← GPR[rt]
CPR[0,rd,sel] ← datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Doubleword Move to Floating Point DMTC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	DMT 00101	rt	fs	0 000 0000 0000
6	5	5	5	11

Format: `DMTC1 rt, fs` **MIPS64**

Purpose:

To copy a doubleword from a GPR to an FPR

Description: `fs← rt`

The doubleword contents of FPR *fs* are placed into FPR *fs*.

Restrictions:

None

Operation:

```
datadoubleword ← GPR[rt]
StoreFPR(fs, UNINTERPRETED_DOUBLEWORD, datadoubleword)
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Doubleword Move to Coprocessor 2 DMTC2

31 26	25 21	20 16	15 11	10 3	2 0
COP2 010010	DMT 00101	rt	rd	0 0 0000 000	sel
6	5	5	5	8	3

Format: `DMTC2 rt,rd` **MIPS64**
`DMTC2 rt, rd, sel` **MIPS64**

Purpose:

To move a doubleword from a GPR to a coprocessor 2 register.

Description: `CPR[2, rd, sel] ← rt`

The contents of GPR *rt* are loaded into the coprocessor 2 register specified by the *rd* and *sel* fields. Note that not all coprocessor 2 registers may support the *sel* field. In those instances, the *sel* field must be zero.

Restrictions:

The results are **UNPREDICTABLE** if coprocessor 2 does not contain a register as specified by *rd* and *sel*, or if the coprocessor 2 register specified by *rd* and *sel* is a 32-bit register.

Operation:

```
datadoubleword ← GPR[rt]
CPR[2,rd,sel]← datadoubleword
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Doubleword Multiply DMULT

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DMULT 011100
6	5	5	10	6

Format: `DMULT rs, rt` **MIPS64**

Purpose:

To multiply 64-bit signed integers

Description: `(LO, HI) ← rs × rt`

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

No arithmetic exception occurs under any circumstances.

Restrictions:

None

Operation:

```
prod← GPR[rs] × GPR[rt]
LO ← $prod_{63..0}$
HI ← $prod_{127..64}$
```

Exceptions:

Reserved Instruction

Doubleword Multiply Unsigned DMULTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DMULTU 011101
6	5	5	10	6

Format: `DMULTU rs, rt` **MIPS64**

Purpose:

To multiply 64-bit unsigned integers

Description: `(LO, HI) ← rs × rt`

The 64-bit doubleword value in GPR *rt* is multiplied by the 64-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 128-bit result. The low-order 64-bit doubleword of the result is placed into special register *LO*, and the high-order 64-bit doubleword is placed into special register *HI*. No arithmetic exception occurs under any circumstances.

Restrictions:

None

Operation:

```
prod← (0||GPR[rs]) × (0||GPR[rt])
LO  ← prod$_{63..0}$
HI  ← prod$_{127..64}$
```

Exceptions:

Reserved Instruction

Doubleword Shift Left Logical DSLL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL 111000
6	5	5	5	5	6

Format: `DSLL rd, rt, sa` **MIPS64**

Purpose:

To execute a left-shift of a doubleword by a fixed amount—0 to 31 bits

Description: `rd ← rt << sa`

The 64-bit doubleword contents of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit-shift amount in the range 0 to 31 is specified by *sa*.

Restrictions:

None

Operation:

```
s      ← 0 || sa
GPR[rd]← GPR[rt]$_{(63–s)..0}$ || 0$^{s}$
```

Exceptions:

Reserved Instruction

Doubleword Shift Left Logical Plus 32 DSLL32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL32 111100
6	5	5	5	5	6

Format: `DSLL32 rd, rt, sa` **MIPS64**

Purpose:

To execute a left-shift of a doubleword by a fixed amount—32 to 63 bits

Description: `rd ← rt << (sa+32)`

The 64-bit doubleword contents of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit-shift amount in the range 0 to 31 is specified by *sa*.

Restrictions:

None

Operation:

```
s          ← 1 || sa    /* 32+sa */
GPR[rd]    ← $\text{GPR[rt]}_{(63-s)..0}$ || $0^{s}$
```

Exceptions:

Reserved Instruction

Doubleword Shift Left Logical Variable DSLLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSLLV 010100
6	5	5	5	5	6

Format: `DSLLV rd, rt, sa` **MIPS64**

Purpose:

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

Description: `rd ← rt << rs`

The 64-bit doubleword contents of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit-shift amount in the range 0 to 63 is specified by the low-order 6 bits in GPR *rs*.

Restrictions:

None

Operation:

```
    s         ← GPR[rs]$_{5..0}$
    GPR[rd]   ← GPR[rt]$_{(63–s)..0}$ || 0$^{s}$
```

Exceptions:

Reserved Instruction

Doubleword Shift Right Arithmetic **DSRA**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA 111011
6	5	5	5	5	6

Format: `DSRA   rd, rt, sa` **MIPS64**

Purpose:

To execute an arithmetic right-shift of a doubleword by a fixed amount—0 to 31 bits

Description: `rd ← rt >> sa      (arithmetic)`

The 64-bit doubleword contents of GPR *rt* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR *rd*. The bit-shift amount in the range 0 to 31 is specified by *sa*.

Restrictions:

None

Operation:

```
s          ← 0 || sa
GPR[rd]    ← $(\text{GPR[rt]}_{63})^{s}$ || $\text{GPR[rt]}_{63..s}$
```

Exceptions:

Reserved Instruction

Doubleword Shift Right Arithmetic Plus 32 — DSRA32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA32 111111
6	5	5	5	5	6

Format: `DSRA32   rd, rt, sa` **MIPS64**

Purpose:

To execute an arithmetic right-shift of a doubleword by a fixed amount—32 to 63 bits

Description: `rd ← rt >> (sa+32)      (arithmetic)`

The doubleword contents of GPR *rt* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR *rd*. The bit-shift amount in the range 32 to 63 is specified by *sa*+32.

Restrictions:

None

Operation:

```
s          ← 1 || sa    /* 32+sa */
GPR[rd]    ← $(GPR[rt]_{63})^{s}$ || $GPR[rt]_{63..s}$
```

Exceptions:

Reserved Instruction

Doubleword Shift Right Arithmetic Variable DSRAV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSRAV 010111
6	5	5	5	5	6

Format: `DSRAV rd, rt, sa` **MIPS64**

Purpose:

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

Description: `rd ← rt >> rs (arithmetic)`

The doubleword contents of GPR *rt* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR *rd*. The bit-shift amount in the range 0 to 63 is specified by the low-order 6 bits in GPR *rs*.

Restrictions:

None

Operation:

```
s          ← GPR[rs]$_{5..0}$
GPR[rd]    ← (GPR[rt]$_{63}$)$^{s}$ || GPR[rt]$_{63..s}$
```

Exceptions:

Reserved Instruction

Doubleword Shift Right Logical — DSRL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRL 111010
6	5	5	5	5	6

Format: `DSRL rd, rt, sa` **MIPS64**

Purpose:

To execute a logical right-shift of a doubleword by a fixed amount—0 to 31 bits

Description: `rd ← rt >> sa (logical)`

The doubleword contents of GPR *rt* are shifted right, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit-shift amount in the range 0 to 31 is specified by *sa*.

Restrictions:

None

Operation:

```
s         ← 0 || sa
GPR[rd]   ← $0^s$ || $GPR[rt]_{63..s}$
```

Exceptions:

Reserved Instruction

Doubleword Shift Right Logical Plus 32 DSRL32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRL32 111110
6	5	5	5	5	6

Format: `DSRL32   rd, rt, sa` **MIPS64**

Purpose:

To execute a logical right-shift of a doubleword by a fixed amount—32 to 63 bits

Description: `rd ← rt >> (sa+32)      (logical)`

The 64-bit doubleword contents of GPR *rt* are shifted right, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit-shift amount in the range 32 to 63 is specified by *sa*+32.

Restrictions:

None

Operation:

```
s         ← 1 || sa    /* 32+sa */
GPR[rd]   ← $0^s$ || $GPR[rt]_{63..s}$
```

Exceptions:

Reserved Instruction

Doubleword Shift Right Logical Variable — DSRLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSRLV 010110
6	5	5	5	5	6

Format: `DSRLV rd, rt, rs` **MIPS64**

Purpose:

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

Description: `rd ← rt >> rs (logical)`

The 64-bit doubleword contents of GPR *rt* are shifted right, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit-shift amount in the range 0 to 63 is specified by the low-order 6 bits in GPR *rs*.

Restrictions:

None

Operation:

```
s          ← $GPR[rs]_{5..0}$
GPR[rd]    ← $0^{s}$ || $GPR[rt]_{63..s}$
```

Exceptions:

Reserved Instruction

Doubleword Subtract DSUB

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSUB 101110
6	5	5	5	5	6

Format: `DSUB   rd, rs, rt` **MIPS64**

Purpose:

To subtract 64-bit integers; trap on overflow

Description: `rd ← rs - rt`

The 64-bit doubleword value in GPR *rt* is subtracted from the 64-bit value in GPR *rs* to produce a 64-bit result. If the subtraction results in 64-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 64-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
temp ← (GPR[rs]$_{63}$||GPR[rs]) – (GPR[rt]$_{63}$||GPR[rt])
if (temp$_{64}$ ≠ temp$_{63}$) then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp$_{63..0}$
endif
```

Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

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

Doubleword Subtract Unsigned DSUBU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSUBU 101111
6	5	5	5	5	6

Format: `DSUBU rd, rs, rt` **MIPS64**

Purpose:

To subtract 64-bit integers

Description: `rd ← rs - rt`

The 64-bit doubleword value in GPR *rt* is subtracted from the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation: 64-bit processors

```
GPR[rd] ← GPR[rs] – GPR[rt]
```

Exceptions:

Reserved Instruction

Programming Notes:

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

Exception Return ERET

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	ERET 011000
6	1	19	6

Format: `ERET` `MIPS32`

Purpose:

To return from interrupt, exception, or error trap.

Description:

ERET returns to the interrupted instruction at the completion of interrupt, exception, or error trap processing. 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 executed in the delay slot of a branch or jump instruction.

An ERET placed between an LL and SC instruction will always cause the SC to fail.

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 addressing mode.

Operation:

```
if $Status_{ERL}$ = 1 then
    temp ← ErrorEPC
    $Status_{ERL}$ ← 0
else
    temp ← EPC
    $Status_{EXL}$ ← 0
endif
if IsMIPS16Implemented() then
    PC ← $temp_{63..1}$ || 0
    ISAMode ← $temp_0$
else
    PC ← temp
endif
LLbit ← 0
```

Exceptions:

Coprocessor Unusable Exception

Floating Point Floor Convert to Long Fixed Point FLOOR.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	FLOOR.L 001011
6	5	5	5	5	6

Format: `FLOOR.L.S fd, fs` **MIPS64**
`FLOOR.L.D fd, fs` **MIPS64**

Purpose:

To convert an FP value to 64-bit fixed point, rounding down

Description: `fd ← convert_and_round(fs)`

The value in FPR *fs*, in format *fmt*, is converted to a value in 64-bit long fixed point format and rounded toward -∞ (rounding mode 3). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation Enable bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}–1$, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *fmt* and *fd* for long fixed point—if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Floating Point Floor Convert to Long Fixed Point (cont.) FLOOR.L.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow

Floating Point Floor Convert to Word Fixed Point FLOOR.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	FLOOR.W 001111
6	5	5	5	5	6

Format: `FLOOR.W.S fd, fs` **MIPS32**
`FLOOR.W.D fd, fs` **MIPS32**

Purpose:

To convert an FP value to 32-bit fixed point, rounding down

Description: `fd ← convert_and_round(fs)`

The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format and rounded toward –∞ (rounding mode 3). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to 2^{31}-1, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, 2^{31}–1, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs—*fs* for type *fmt* and *fd* for word fixed point—if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact, Overflow

Jump J

31 26	25 0
J 000010	instr_index
6	26

Format: `J target` **MIPS32**

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:

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

Operation:

```
I:
I+1:PC ← $PC_{GPRLEN-1..28}$ || instr_index || $0^2$
```

Exceptions:

None

Programming Notes:

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

This definition creates the following boundary case: When the jump 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 000011	instr_index
6	26

Format: `JAL target` **MIPS32**

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:

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

Operation:

```
I:   GPR[31]← PC + 8
I+1:PC    ← $PC_{GPRLEN-1..28}$ || instr_index || $0^2$
```

Exceptions:

None

Programming Notes:

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

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

Jump and Link Register JALR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	0 00000	rd	hint	JALR 001001
6	5	5	5	5	6

Format: `JALR rs (rd = 31 implied)` **MIPS32**
`JALR rd, rs` **MIPS32**

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.

For processors that do not implement the MIPS16 ASE:

- 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.

For processors that do implement the MIPS16 ASE:

- Jump to the effective target address in GPR *rs*. Set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

At this time the only defined hint field value is 0, which sets default handling of JALR. Future versions of the architecture 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. For processors that do not implement the MIPS16 ASE, 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. For processors that do implement the MIPS16 ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

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

Jump and Link Register, cont. JALR

Operation:

```
I:  temp ← GPR[rs]
    GPR[rd] ← PC + 8
I+1:if $Config1_{CA}$ = 0 then
        PC ← temp
    else
        PC ← $temp_{GPRLEN-1..1}$ || 0
        ISAMode ← $temp_0$
    endif
```

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	0 00 0000 0000	hint	JR 001000
6	5	10	5	6

Format: `JR rs` **MIPS32**

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.

For processors that implement the MIPS16 ASE, set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

Restrictions:

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16 ASE, 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. For processors that do implement the MIPS16 ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

At this time the only defined hint field value is 0, which sets default handling of JR. Future versions of the architecture may define additional hint values.

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

Operation:

```
I:   temp ← GPR[rs]
I+1:if Config1$_{CA}$ = 0 then
        PC ← temp
    else
        PC ← temp$_{GPRLEN-1..1}$ || 0
        ISAMode ← temp$_0$
    endif
```

Exceptions:

None

Jump Register, cont. JR

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 100000	base	rt	offset
6	5	5	16

Format: `LB rt, offset(base)` **MIPS32**

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

Exceptions:

TLB Refill, TLB Invalid, Address Error

Load Byte Unsigned LBU

31 26	25 21	20 16	15 0
LBU 100100	base	rt	offset
6	5	5	16

Format: `LBU rt, offset(base)` **MIPS32**

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$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor ReverseEndian$^3$)
memdoubleword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte   ← vAddr$_{2..0}$ xor BigEndianCPU$^3$
GPR[rt]← zero_extend(memdoubleword$_{7+8*byte..8*byte}$)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error

Load Doubleword LD

31 26	25 21	20 16	15 0
LD 110111	base	rt	offset
6	5	5	16

Format: `LD rt, offset(base)` **MIPS64**

Purpose:

To load a doubleword from memory

Description: `rt ← memory[base+offset]`

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched 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 any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr          ← sign_extend(offset) + GPR[base]
if $vAddr_{2..0} \neq 0^3$ then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[rt]        ← memdoubleword
```

Exceptions:

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

Load Doubleword to Floating Point LDC1

31 26	25 21	20 16	15 0
LDC1 110101	base	ft	offset
6	5	5	16

Format: `LDC1 ft, offset(base)` **MIPS32**

Purpose:

To load a doubleword from memory to an FPR

Description: `ft ← memory[base+offset]`

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in FPR *ft*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

An Address Error exception occurs if EffectiveAddress$_{2..0}$ ≠ 0 (not doubleword-aligned).

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr$_{2..0}$ ≠ $0^3$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
StoreFPR(ft, UNINTERPRETED_DOUBLEWORD, memdoubleword)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, Address Error

Load Doubleword to Coprocessor 2 LDC2

31 26	25 21	20 16	15 0
LDC2 110110	base	rt	offset
6	5	5	16

Format: `LDC2 rt, offset(base)` **MIPS32**

Purpose:

To load a doubleword from memory to a Coprocessor 2 register

Description: `rt ← memory[base+offset]`

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in Coprocessor 2 register *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

An Address Error exception occurs if EffectiveAddress$_{2..0}$ ≠ 0 (not doubleword-aligned).

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if $vAddr_{2..0} \ne 0^3$ then SignalException(AddressError) endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
CPR[2,rt,0] ← memdoubleword
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, Address Error

Load Doubleword Left **LDL**

31 26	25 21	20 16	15 0
LDL 011010	base	rt	offset
6	5	5	16

Format: `LDL rt, offset(base)` **MIPS64**

Purpose:

To load the most-significant part of a doubleword 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 8 consecutive bytes forming a doubleword *(DW*) in memory, starting at an arbitrary byte boundary.

A part of *DW*, the most-significant 1 to 8 bytes, is in the aligned doubleword containing *EffAddr.* This part of *DW* is loaded appropriately into the most-significant (left) part of GPR *rt*, leaving the remainder of GPR *rt* unchanged.

Figure 12-5 Unaligned Doubleword Load Using LDL and LDR

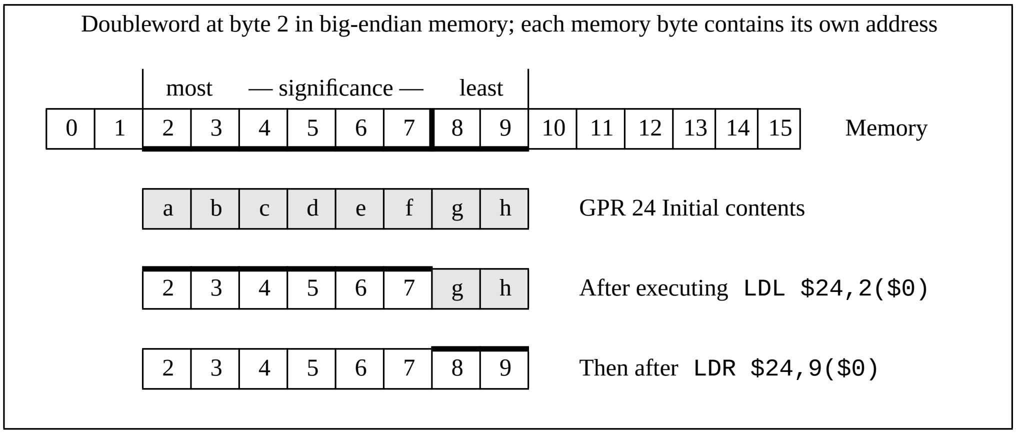
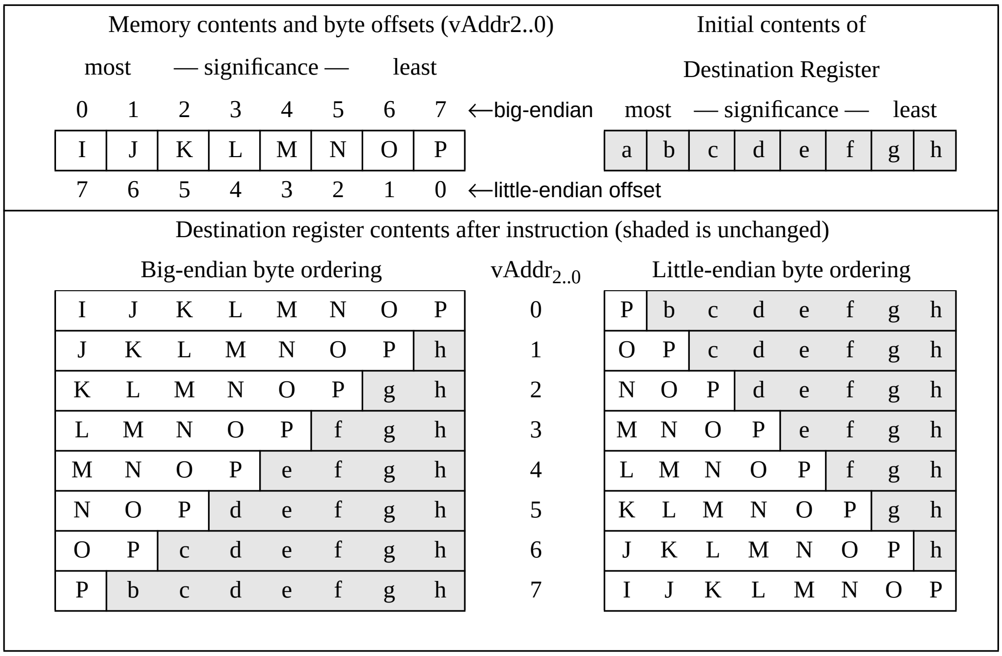

Doubleword at byte 2 in big-endian memory; each memory byte contains its own address
most — significance — least
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Memory
a b c d e f g h
GPR 24 Initial contents
2 3 4 5 6 7 g h
After executing LDL $24,2($0)
2 3 4 5 6 7 8 9
Then after LDR $24,9($0)

Figure 12-5 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 6 bytes, is located in the aligned doubleword starting with the most-significant byte at 2. LDL first loads these 6 bytes into the left part of the destination register and leaves the remainder of the destination unchanged. The complementary LDR next loads the remainder of the unaligned doubleword.

Load Doubleword Left (cont.) LDL

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned doubleword—the low 3 bits of the address (*vAddr2..0*)—and the current byte-ordering mode of the processor (big- or little-endian). Figure 12-6 shows the bytes loaded for every combination of offset and byte ordering.

Figure 12-6 Bytes Loaded by LDL Instruction



Memory contents and byte offsets (vAddr2..0)

most — significance — least

0	1	2	3	4	5	6	7	←big-endian
I	J	K	L	M	N	O	P	
7	6	5	4	3	2	1	0	←little-endian offset

Initial contents of Destination Register

most — significance — least

a	b	c	d	e	f	g	h

Destination register contents after instruction (shaded is unchanged)

Big-endian byte ordering								$vAddr_{2..0}$	Little-endian byte ordering							
I	J	K	L	M	N	O	P	0	P	b	c	d	e	f	g	h
J	K	L	M	N	O	P	h	1	O	P	c	d	e	f	g	h
K	L	M	N	O	P	g	h	2	N	O	P	d	e	f	g	h
L	M	N	O	P	f	g	h	3	M	N	O	P	e	f	g	h
M	N	O	P	e	f	g	h	4	L	M	N	O	P	f	g	h
N	O	P	d	e	f	g	h	5	K	L	M	N	O	P	g	h
O	P	c	d	e	f	g	h	6	J	K	L	M	N	O	P	h
P	b	c	d	e	f	g	h	7	I	J	K	L	M	N	O	P

Restrictions:

None

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor ReverseEndian$^3$)
if BigEndianMem = 0 then
   pAddr ← pAddr$_{PSIZE-1..3}$ || 0$^3$
endif
byte   ← vAddr$_{2..0}$ xor BigEndianCPU$^3$
memdoubleword← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
GPR[rt]← memdoublworde$_{7+8*byte..0}$ || GPR[rt]$_{55–8*byte..0}$
```

Exceptions:

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

Load Doubleword Right LDR

31 26	25 21	20 16	15 0
LDR 011011	base	rt	offset
6	5	5	16

Format: `LDR rt, offset(base)` **MIPS64**

Purpose:

To load the least-significant part of a doubleword from an unaligned memory address

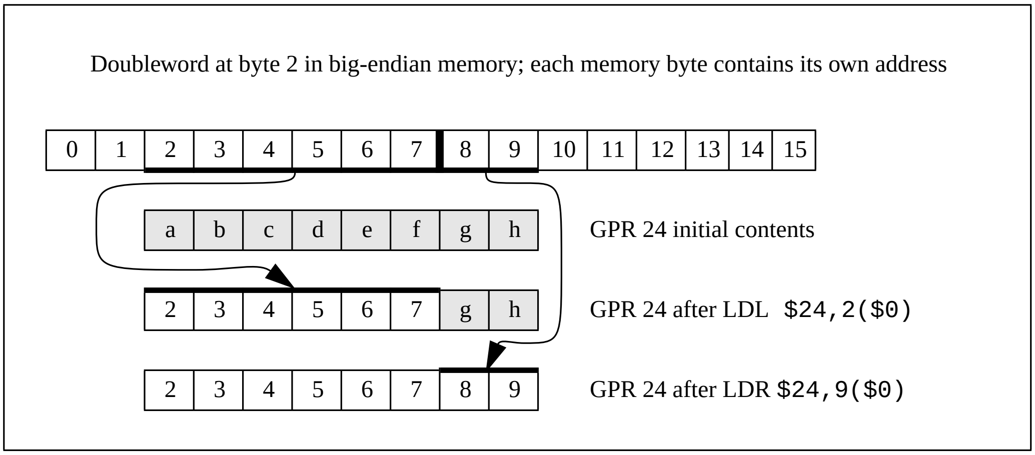
Description: `rt ← rt MERGE memory[base+offset]`

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 least-significant of 8 consecutive bytes forming a doubleword *(DW*) in memory, starting at an arbitrary byte boundary.

A part of *DW*, the least-significant 1 to 8 bytes, is in the aligned doubleword containing *EffAddr.* This part of *DW* is loaded appropriately into the least-significant (right) part of GPR *rt* leaving the remainder of GPR *rt* unchanged.

Figure 12-7 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. Two bytes of the *DW* are located in the aligned doubleword containing the least-significant byte at 9. LDR first loads these 2 bytes into the right part of the destination register, and leaves the remainder of the destination unchanged. The complementary LDL next loads the remainder of the unaligned doubleword.

Figure 12-7 Unaligned Doubleword Load Using LDR and LDL

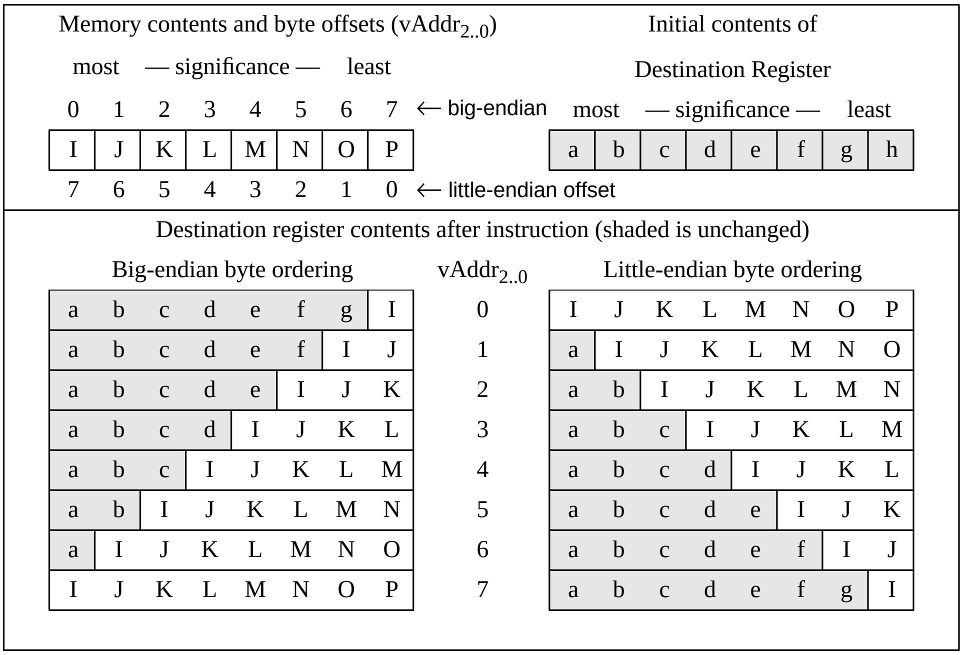

Doubleword at byte 2 in big-endian memory; each memory byte contains its own address
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a b c d e f g h
GPR 24 initial contents
2 3 4 5 6 7 g h
GPR 24 after LDL $24,2($0)
2 3 4 5 6 7 8 9
GPR 24 after LDR $24,9($0)

Load Doubleword Right (cont.) LDR

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned doubleword—the low 3 bits of the address (*vAddr2..0)*—and the current byte-ordering mode of the processor (big- or little-endian).

Figure 12-8 shows the bytes loaded for every combination of offset and byte ordering.

Figure 12-8 Bytes Loaded by LDR Instruction



Memory contents and byte offsets ($vAddr_{2..0}$)

most — significance — least

0	1	2	3	4	5	6	7	← big-endian
I	J	K	L	M	N	O	P	
7	6	5	4	3	2	1	0	← little-endian offset

Initial contents of

Destination Register

most — significance — least

a	b	c	d	e	f	g	h

Destination register contents after instruction (shaded is unchanged)

Big-endian byte ordering	$vAddr_{2..0}$	Little-endian byte ordering
a b c d e f g I	0	I J K L M N O P
a b c d e f I J	1	a I J K L M N O
a b c d e I J K	2	a b I J K L M N
a b c d I J K L	3	a b c I J K L M
a b c I J K L M	4	a b c d I J K L
a b I J K L M N	5	a b c d e I J K
a I J K L M N O	6	a b c d e f I J
I J K L M N O P	7	a b c d e f g I

Restrictions:

None

Load Doubleword Right (cont.) LDR

Operation: 64-bit processors

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$  xor $ReverseEndian^{3}$)
if BigEndianMem = 1 then
   pAddr ← $pAddr_{PSIZE-1..3}$ || $0^{3}$
endif
byte   ← $vAddr_{2..0}$ xor $BigEndianCPU^{3}$
memdoubleword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
GPR[rt] ← $GPR[rt]_{63..64-8*byte}$ || $memdoubleword_{63..8*byte}$
```

Exceptions:

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

Load Doubleword Indexed to Floating Point LDXC1

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	0 00000	fd	LDXC1 000001
6	5	5	5	5	6

Format: `LDXC1 fd, index(base)` **MIPS64**

Purpose:

To load a doubleword from memory to an FPR (GPR+GPR addressing)

Description: `fd ← memory[base+index]`

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in FPR *fd*. The contents of GPR *index* and GPR *base* are added to form the effective address.

Restrictions:

An Address Error exception occurs if $\text{EffectiveAddress}_{2..0} \neq 0$ (not doubleword-aligned).

Operation:

```
vAddr ← GPR[base] + GPR[index]
if $vAddr_{2..0} \neq 0^3$ then SignalException(AddressError) endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
StoreFPR (fd, UNINTERPRETED_DOUBLEWORD, memdoubleword)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable

Load Halfword LH

31 26	25 21	20 16	15 0
LH 100001	base	rt	offset
6	5	5	16

Format: `LH rt, offset(base)` **MIPS32**

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 $vAddr_0 \neq 0$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor ($ReverseEndian^2$ || 0))
memdoubleword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte   ← $vAddr_{2..0}$ xor ($BigEndianCPU^2$ || 0)
GPR[rt] ← sign_extend($memdoubleword_{15+8*byte..8*byte}$)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Load Halfword Unsigned — LHU

31 26	25 21	20 16	15 0
LHU 100101	base	rt	offset
6	5	5	16

Format: `LHU rt, offset(base)` **MIPS32**

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 $vAddr_0 \neq 0$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor ($ReverseEndian^2$ || 0))
memdoubleword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte   ← $vAddr_{2..0}$ xor ($BigEndianCPU^2$ || 0)
GPR[rt] ← zero_extend($memdoubleword_{15+8*byte..8*byte}$)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error

Load Linked Word LL

31 26	25 21	20 16	15 0
LL 110000	base	rt	offset
6	5	5	16

Format: `LL rt, offset(base)` **MIPS32**

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.

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 $vAddr_{1..0} \ne 0^2$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || $0^2$))
memdoubleword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← $vAddr_{2..0}$ xor (BigEndianCPU || $0^2$)
GPR[rt] ← sign_extend($memdoubleword_{31+8*byte..8*byte}$)
LLbit  ← 1
```

Load Linked Word (cont.) LL

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 Linked Doubleword LLD

31 26	25 21	20 16	15 0
LLD 110100	base	rt	offset
6	5	5	16

Format: `LLD rt, offset(base)` **MIPS64**

Purpose:

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

Description: `rt ← memory[base+offset]`

The LLD and SCD 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 contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched 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 LLD is executed it starts the active RMW sequence and replaces any other sequence that was active.

The RMW sequence is completed by a subsequent SCD instruction that either completes the RMW sequence atomically and succeeds, or does not complete and fails.

Executing LLD on one processor does not cause an action that, by itself, would cause an SCD for the same block to fail on another processor.

An execution of LLD does not have to be followed by execution of SCD; a program is free to abandon the RMW sequence without attempting a write.

Restrictions:

The addressed location must be cached; if it is not, the result is undefined.

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

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if $vAddr_{2..0} \ne 0^3$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memdoubleword ← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[rt] ← memdoubleword
LLbit  ← 1
```

Load Linked Doubleword (cont.) LLD

Exceptions:

TLB Refill, TLB Invalid, Address Error, Reserved Instruction

Load Upper Immediate LUI

31 26	25 21	20 16	15 0
LUI 001111	0 00000	rt	immediate
6	5	5	16

Format: `LUI rt, immediate` **MIPS32**

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 Doubleword Indexed Unaligned to Floating Point LUXC1

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	0 00000	fd	LUXC1 000101
6	5	5	5	5	6

Format: `LUXC1 fd, index(base)` **MIPS64**

Purpose:

To load a doubleword from memory to an FPR (GPR+GPR addressing), ignoring alignment

Description: fd ← memory[(base+index)$_{\text{PSIZE-1..3}}$]

The contents of the 64-bit doubleword at the memory location specified by the effective address are fetched and placed into the low word of coprocessor 1 general register *fd*. The contents of GPR *index* and GPR *base* are added to form the effective address. The effective address is doubleword-aligned; EffectiveAddress$_{2..0}$ are ignored.

Restrictions:

The result of this instruction is undefined if the processor is executing in 16 FP registers mode.

Operation:

```
vAddr  ← (GPR[base]+GPR[index])$_{63..3}$ || $0^3$
(pAddr, CCA) ← AddressTranslation(vaddr, DATA, LOAD)
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
StoreFPR(ft, UNINTERPRETED, memdoubleword)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified

Load Word LW

31 26	25 21	20 16	15 0
LW 100011	base	rt	offset
6	5	5	16

Format: `LW rt, offset(base)` **MIPS32**

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 vAddr$_{1..0}$ ≠ 0$^2$ then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor (ReverseEndian || 0$^2$))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← vAddr$_{2..0}$ xor (BigEndianCPU || 0$^2$)
GPR[rt]← sign_extend(memdoubleword$_{31+8*byte..8*byte}$)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Load Word to Floating Point LWC1

31 26	25 21	20 16	15 0
LWC1 110001	base	rt	offset
6	5	5	16

Format: `LWC1 ft, offset(base)` **MIPS32**

Purpose:

To load a word from memory to an FPR

Description: `ft ← memory[base+offset]`

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of coprocessor 1 general register *ft*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

An Address Error exception occurs if EffectiveAddress$_{1..0}$ ≠ 0 (not word-aligned).

Operation:

```
/* mem is aligned 64 bits from memory.  Pick out correct bytes. */
vAddr ← sign_extend(offset) + GPR[base]
if vAddr$_{1..0}$ ≠ $0^2$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor (ReverseEndian || $0^2$))
memdoubleword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
bytesel ← vAddr$_{2..0}$ xor (BigEndianCPU || $0^2$)
StoreFPR(ft, UNINTERPRETED_WORD,
          sign_extend(memdoubleword$_{31+8*bytesel..8*bytesel}$))
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable

Load Word to Coprocessor 2 LWC2

31 26	25 21	20 16	15 0
LWC2 110010	base	rt	offset
6	5	5	16

Format: `LWC2 rt, offset(base)` **MIPS32**

Purpose:

To load a word from memory to a COP2 register

Description: `rt ← memory[base+offset]`

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of COP2 (Coprocessor 2) general register *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

An Address Error exception occurs if EffectiveAddress$_{1..0}$ ≠ 0 (not word-aligned).

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr$_{12..0}$ ≠ $0^2$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor (ReverseEndian || $0^2$))
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
bytesel ← vAddr$_{2..0}$ xor (BigEndianCPU || $0^2$)
CPR[2,rt,0] ← sign_extend(memdoubleword$_{31+8*bytesel..8*bytesel}$)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable

Load Word Left LWL

31 26	25 21	20 16	15 0
LWL 100010	base	rt	offset
6	5	5	16

Format: `LWL rt, offset(base)` **MIPS32**

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]`

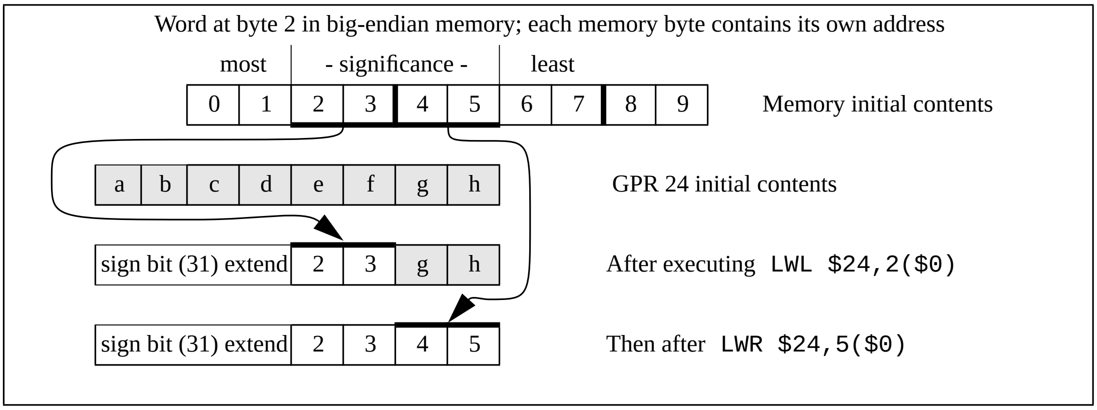
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

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.

For 64-bit GPR *rt* registers, the destination word is the low-order word of the register. The loaded value is treated as a signed value; the word sign bit (bit 31) is always loaded from memory and the new sign bit value is copied into bits 63..32.

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

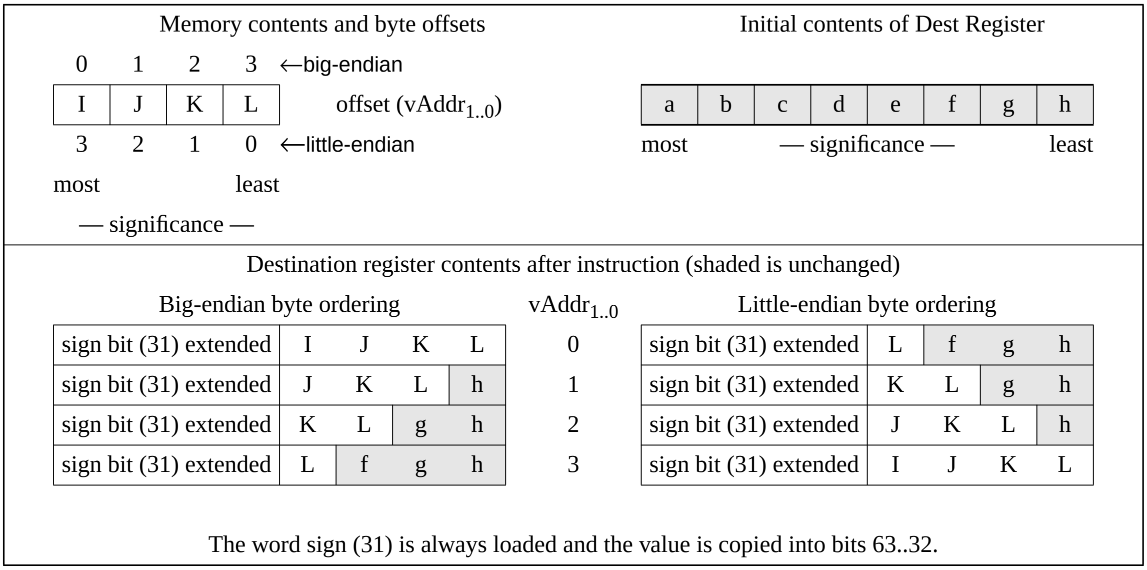
Figure 12-9 Unaligned Word Load Using LWL and LWR



Load Word Left (con't) 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). The figure below shows the bytes loaded for every combination of offset and byte ordering.

Figure 12-10 Bytes Loaded by LWL Instruction



Memory contents and byte offsets

0	1	2	3	←big-endian
I	J	K	L	offset ($vAddr_{1..0}$)
3	2	1	0	←little-endian
most			least	

— significance —

Initial contents of Dest Register

a	b	c	d	e	f	g	h
most		— significance —					least

Destination register contents after instruction (shaded is unchanged)

Big-endian byte ordering					$vAddr_{1..0}$	Little-endian byte ordering				
sign bit (31) extended	I	J	K	L	0	sign bit (31) extended	L	f	g	h
sign bit (31) extended	J	K	L	h	1	sign bit (31) extended	K	L	g	h
sign bit (31) extended	K	L	g	h	2	sign bit (31) extended	J	K	L	h
sign bit (31) extended	L	f	g	h	3	sign bit (31) extended	I	J	K	L

The word sign (31) is always loaded and the value is copied into bits 63..32.

Load Word Left (con’t) LWL

Restrictions:

None

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor ReverseEndian$^{3}$)
if BigEndianMem = 0 then
   pAddr← pAddr$_{PSIZE-1..3}$ || 0$^{3}$
endif
byte   ← 0 || (vAddr$_{1..0}$ xor BigEndianCPU$^{2}$)
word   ← vAddr$_{2}$ xor BigEndianCPU
memdoubleword← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp   ← memdoubleword$_{31+32*word-8*byte..32*word}$ || GPR[rt]$_{23-8*byte..0}$
GPR[rt]← (temp$_{31}$)$^{32}$ || temp
```

Exceptions:

None

TLB Refill, TLB Invalid, Bus Error, Address Error

Programming Notes:

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

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)` **MIPS32**

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.

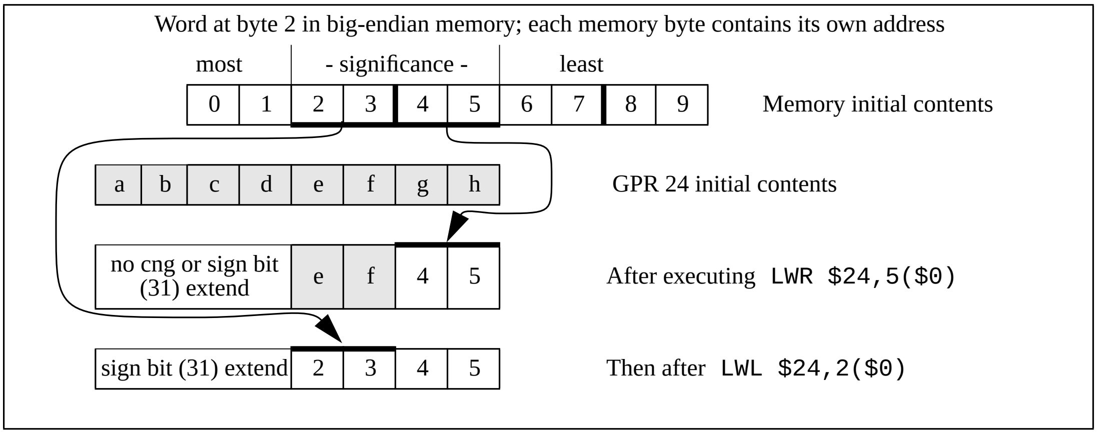
If GPR *rt* is a 64-bit register, the destination word is the low-order word of the register. The loaded value is treated as a signed value; if the word sign bit (bit 31) is loaded (that is, when all 4 bytes are loaded), then the new sign bit value is copied into bits 63..32. If bit 31 is not loaded, the value of bits 63..32 is implementation dependent; the value is either unchanged or a copy of the current value of bit 31.

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

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

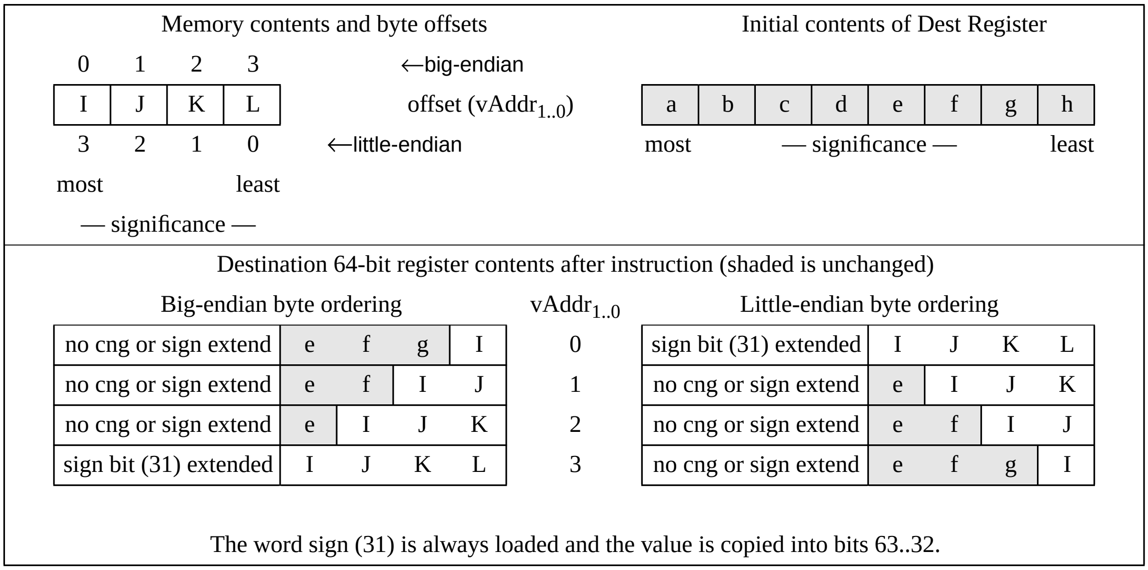
Figure 12-11 Unaligned Word Load Using LWL and LWR



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

Load Word Right (cont.) LWR

Figure 12-12 Bytes Loaded by LWL Instruction



Memory contents and byte offsets

0	1	2	3	←big-endian
I	J	K	L	offset ($vAddr_{1..0}$)
3	2	1	0	←little-endian

most — significance — least

Initial contents of Dest Register

a	b	c	d	e	f	g	h

most — significance — least

Destination 64-bit register contents after instruction (shaded is unchanged)

Big-endian byte ordering					$vAddr_{1..0}$	Little-endian byte ordering				
no cng or sign extend	e	f	g	I	0	sign bit (31) extended	I	J	K	L
no cng or sign extend	e	f	I	J	1	no cng or sign extend	e	I	J	K
no cng or sign extend	e	I	J	K	2	no cng or sign extend	e	f	I	J
sign bit (31) extended	I	J	K	L	3	no cng or sign extend	e	f	g	I

The word sign (31) is always loaded and the value is copied into bits 63..32.

Load Word Right (cont.) LWR

Restrictions:

None

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor $ReverseEndian^3$)
if BigEndianMem = 0 then
   pAddr← $pAddr_{PSIZE-1..3}$ || $0^3$
endif
byte   ← $vAddr_{1..0}$ xor $BigEndianCPU^2$
word   ← $vAddr_2$ xor BigEndianCPU
memdoubleword← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp   ← $GPR[rt]_{31..32-8*byte}$ || $memdoubleword_{31+32*word..32*word+8*byte}$
if byte = 4 then
   utemp← $(temp_{31})^{32}$/* loaded bit 31, must sign extend */
else
            /* one of the following two behaviors: */
   utemp← $GPR[rt]_{63..32}$       /* leave what was there alone */
   utemp← $(GPR[rt]_{31})^{32}$       /* sign-extend bit 31 */
endif
GPR[rt]← utemp || temp
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Programming Notes:

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

Load Word Unsigned LWU

31 26	25 21	20 16	15 0
LWU 100111	base	rt	offset
6	5	5	16

Format: `LWU rt, offset(base)` **MIPS64**

Purpose:

To load a word from memory as an unsigned 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, 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 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 $vAddr_{1..0} \neq 0^2$ then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || $0^2$))
memdoubleword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
byte   ← $vAddr_{2..0}$ xor (BigEndianCPU || $0^2$)
GPR[rt]← $0^{32}$ ∥ $memdoubleword_{31+8*byte..8*byte}$
```

Exceptions:

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

Load Word Indexed to Floating Point LWXC1

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	0 00000	fd	LWXC1 000000
6	5	5	5	5	6

Format: `LWXC1 fd, index(base)` **MIPS64**

Purpose:

To load a word from memory to an FPR (GPR+GPR addressing)

Description: `fd ← memory[base+index]`

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of coprocessor 1 general register *fd*. The contents of GPR *index* and GPR *base* are added to form the effective address.

Restrictions:

An Address Error exception occurs if $\text{EffectiveAddress}_{1..0} \neq 0$ (not word-aligned).

Operation:

```
vAddr ← GPR[base] + GPR[index]
if $vAddr_{1..0} \neq 0^2$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || $0^2$))
memdoubleword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
bytesel ← $vAddr_{2..0}$ xor (BigEndianCPU || $0^2$)
StoreFPR(ft, UNINTERPRETED_DOUBLEWORD,
         sign_extend($memdoubleword_{31+8*bytesel..8*bytesel}$))
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable

Multiply and Add Word to Hi,Lo MADD

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	0 0000	0 00000	MADD 000000
6	5	5	5	5	6

Format: `MADD rs, rt` **MIPS32**

Purpose:

To multiply two words and add the result to Hi, Lo

Description: `(HI,LO) ← (HI,LO) + (rs × rt)`

The 32-bit word value in GPR *rs* is multiplied by the 32-bit word 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 $HI_{31..0}$ and $LO_{31..0}$.. The most significant 32 bits of the result are sign-extended and written into *HI* and the least signficant 32 bits are sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← ($HI_{31..0}$ || $LO_{31..0}$) + ($GPR[rs]_{31..0}$ × $GPR[rt]_{31..0}$)
HI ← sign_extend($temp_{63..32}$)
LO ← sign_extend($temp_{31..0}$)
```

Exceptions:

None

Programming Notes:

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

Floating Point Multiply Add **MADD.fmt**

31 26	25 21	20 16	15 11	10 6	5 3	2 0
COP1X 010011	fr	ft	fs	fd	MADD 100	fmt
6	5	5	5	5	3	3

Format: `MADD.S fd, fr, fs, ft` **MIPS64**
`MADD.D fd, fr, fs, ft` **MIPS64**

Purpose:

To perform a combined multiply-then-add of FP values

Description: `fd ← (fs × ft) + fr`

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce an intermediate product. The value in FPR *fr* is added to the product. The result sum is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*.

Cause bits are ORed into the *Flag* bits if no exception is taken.

Restrictions:

The fields *fr*, *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, (vfs $\times_{fmt}$ vft) $+_{fmt}$ vfr)
```

Floating Point Multiply Add (cont.) MADD.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

Multiply and Add Unsigned Word to Hi,Lo MADDU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	0 00000	0 00000	MADDU 000001
6	5	5	5	5	6

Format: `MADDU rs, rt` **MIPS32**

Purpose:

To multiply two unsigned words and add the result to Hi, Lo.

Description: `(HI,LO) ← (HI,LO) + (rs × rt)`

The 32-bit word value in GPR *rs* is multiplied by the 32-bit word 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 $HI_{31..0}$ and $LO_{31..0}$.. The most significant 32 bits of the result are sign-extended and written into *HI* and the least signficant 32 bits are sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← $(HI_{31..0} \| LO_{31..0}) + ((0^{32} \| GPR[rs]_{31..0}) \times (0^{32} \| GPR[rt]_{31..0}))$
HI ← sign_extend($temp_{63..32}$)
LO ← sign_extend($temp_{31..0}$)
```

Exceptions:

None

Programming Notes:

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

Move from Coprocessor 0 MFC0

31 26	25 21	20 16	15 11	10 3	2 0
COP0 010000	MF 00000	rt	rd	0 00000000	sel
6	5	5	5	8	3

Format: `MFC0 rt, rd` **MIPS32**
`MFC0 rt, rd, sel` **MIPS32**

Purpose:

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

Description: `rt ← CPR[0,rd,sel]`

The contents of the coprocessor 0 register specified by the combination of rd and sel are sign-extended and loaded into general register rt. Note that not all coprocessor 0 registers support the sel field. In those instances, the sel field must be zero.

Restrictions:

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

Operation:

```
data ← CPR[0,rd,sel]$_{31..0}$
GPR[rt] ← sign_extend(data)
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Move Word From Floating Point MFC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	MF 00000	rt	fs	0 000 0000 0000
6	5	5	5	11

Format: `MFC1 rt, fs` **MIPS32**

Purpose:

To copy a word from an FPU (CP1) general register to a GPR

Description: `rt ← fs`

The contents of FPR fs are sign-extended and loaded into general register rt.

Restrictions:

None

Operation:

```
data ← ValueFPR(fs, UNINTERPRETED_WORD)$_{31..0}$
GPR[rt] ← sign_extend(data)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Move Word From Coprocessor 2 MFC2

31 26	25 21	20 16	15 11	10 3	2 0
COP2 010010	MF 00000	rt	rd	0 000 0000 0	sel
6	5	5	5	8	3

Format:		
	`MFC2 rt, rd`	**MIPS32**
	`MFC2, rt, rd, sel`	**MIPS32**

Purpose:

To copy a word from a COP2 general register to a GPR

Description: `rt ← CPR[2,rd,sel]`

The contents of the lower 32-bits of GPR *rt* are sign-extended and placed into the coprocessor 2 register specified by the *rd* and *sel* fields. Note that not all coprocessor 2 registers may support the *sel* field. In those instances, the *sel* field must be zero.

Restrictions:

The results are **UNPREDICTABLE** if coprocessor 2 does not contain a register as specified by *rd* and *sel*.

Operation:

```
data ← CPR[2,rd,sel]$_{31..0}$
GPR[rt] ← sign_extend(data)
```

Exceptions:

Coprocessor Unusable

Move From HI Register — MFHI

31 26	25 16	15 11	10 6	5 0
SPECIAL 000000	0 00 0000 0000	rd	0 00000	MFHI 010000
6	10	5	5	6

Format: `MFHI rd` **MIPS32**

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 000000	0 00 0000 0000	rd	0 00000	MFLO 010010
6	10	5	5	6

Format: `MFLO rd` **MIPS32**

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

Floating Point Move MOV.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	MOV 000110
6	5	5	5	5	6

Format: `MOV.S fd, fs` **MIPS32**
`MOV.D fd, fs`**MIPS32**
Purpose:

To move an FP value between FPRs

Description: `fd ← fs`

The value in FPR *fs* is placed into FPR *fd*. The source and destination are values in format *fmt*.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

```
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Move Conditional on Floating Point False MOVF

31 26	25 21	20 18	17	16	15 11	10 6	5 0
SPECIAL 000000	rs	cc	0 0	tf 0	rd	0 00000	MOVCI 000001
6	5	3	1	1	5	5	6

Format: `MOVF rd, rs, cc` **MIPS32**

Purpose:

To test an FP condition code then conditionally move a GPR

Description: `if cc = 0 then rd ← rs`

If the floating point condition code specified by *CC* is zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

None

Operation:

```
if FPConditionCode(cc) = 0 then
   GPR[rd] ← GPR[rs]
endif
```

Exceptions:

Reserved Instruction, Coprocessor Unusable

Floating Point Move Conditional on Floating Point False MOVF.fmt

31 26	25 21	20 18	17	16	15 11	10 6	5 0
COP1 010001	fmt	cc	0 0	tf 0	fs	fd	MOVCF 010001
6	5	3	1	1	5	5	6

Format: `MOVF.S fd, fs, cc` **MIPS32**
`MOVF.D fd, fs, cc` **MIPS32**
Purpose:

To test an FP condition code then conditionally move an FP value

Description: `if cc = 0 then fd ← fs`

If the floating point condition code specified by *CC* is zero, then the value in FPR *fs* is placed into FPR *fd*. The source and destination are values in format *fmt*.

If the condition code is not zero, then FPR *fs* is not copied and FPR *fd* retains its previous value in format *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes **UNPREDICTABLE**.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**. The operand must be a value in format *fmt*; if it is not, the result is **UNPREDITABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Floating Point Move Conditional on Floating Point False (cont.) MOVF.fmt

Operation:

```
if fmt ≠ PS
    if FPConditionCode(cc) = 0 then
        StoreFPR(fd, fmt, ValueFPR(fs, fmt))
    else
        StoreFPR(fd, fmt, ValueFPR(fd, fmt))
    endif
else
    mask ← 0
    if FPConditionCode(cc+0) = 0 then mask ← mask or 0xF0 endif
    if FPConditionCode(cc+1) = 0 then mask ← mask or 0x0F endif
    StoreFPR(fd, PS, ByteMerge(mask, fd, fs))
endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions*:*

Unimplemented Operation

Move Conditional on Not Zero MOVN

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	MOVN 001011
6	5	5	5	5	6

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

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:

None

Programming Notes:

The non-zero value tested here is the *condition true* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

Floating Point Move Conditional on Not Zero **MOVN.fmt**

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	rt	fs	fd	MOVN 010011
6	5	5	5	5	6

Format: `MOVN.S fd, fs, rt` **MIPS32**
`MOVN.D fd, fs, rt` **MIPS32**

Purpose:

To test a GPR then conditionally move an FP value

Description: `if rt ≠ 0 then fd ← fs`

If the value in GPR *rt* is not equal to zero, then the value in FPR *fs* is placed in FPR *fd*. The source and destination are values in format *fmt*.

If GPR *rt* contains zero, then FPR *fs* is not copied and FPR *fd* contains its previous value in format *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes **UNPREDICTABLE**.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Floating Point Move Conditional on Not Zero MOVN.fmt

Operation:

```
if GPR[rt] ≠ 0 then
   StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
   StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions*:*

Unimplemented Operation

Move Conditional on Floating Point True **MOVT**

31 26	25 21	20 18	17	16	15 11	10 6	5 0
SPECIAL	rs	cc	0	tf	rd	0	MOVCI
000000			0	1		00000	000001
6	5	3	1	1	5	5	6

Format: `MOVT rd, rs, cc` **MIPS32**

Purpose:

To test an FP condition code then conditionally move a GPR

Description: `if cc = 1 then rd ← rs`

If the floating point condition code specified by *CC* is one, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

None

Operation:

```
if FPConditionCode(cc) = 1 then
   GPR[rd] ← GPR[rs]
endif
```

Exceptions:

Reserved Instruction, Coprocessor Unusable

Floating Point Move Conditional on Floating Point True **MOVT.fmt**

31 26	25 21	20 18	17	16	15 11	10 6	5 0
COP1 010001	fmt	cc	0 0	tf 1	fs	fd	MOVCF 010001
6	5	3	1	1	5	5	6

Format: `MOVT.S fd, fs, cc` **MIPS32**
`MOVT.D fd, fs, cc` **MIPS32**

Purpose:

To test an FP condition code then conditionally move an FP value

Description: `if cc = 1 then fd ← fs`

If the floating point condition code specified by *CC* is one, then the value in FPR *fs* is placed into FPR *fd*. The source and destination are values in format *fmt*.

If the condition code is not one, then FPR *fs* is not copied and FPR *fd* contains its previous value in format *fm*t. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes undefined.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**. The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Floating Point Move Conditional on Floating Point True **MOVT.fmt**

Operation:

```
if fmt ≠ PS
   if FPConditionCode(cc) = 0 then
      StoreFPR(fd, fmt, ValueFPR(fs, fmt))
   else
      StoreFPR(fd, fmt, ValueFPR(fd, fmt))
   endif
else
   mask ← 0
   if FPConditionCode(cc+0) = 0 then mask ← mask or 0xF0 endif
   if FPConditionCode(cc+1) = 0 then mask ← mask or 0x0F endif
   StoreFPR(fd, PS, ByteMerge(mask, fd, fs))
endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Move Conditional on Zero **MOVZ**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	MOVZ 001010
6	5	5	5	5	6

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

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:

None

Programming Notes:

The zero value tested here is the *condition false* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

Floating Point Move Conditional on Zero MOVZ.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	rt	fs	fd	MOVZ 010010
6	5	5	5	5	6

Format: `MOVZ.S fd, fs, rt` **MIPS32**
`MOVZ.D fd, fs, rt` **MIPS32**

Purpose:

To test a GPR then conditionally move an FP value

Description: `if rt = 0 then fd ← fs`

If the value in GPR *rt* is equal to zero then the value in FPR *fs* is placed in FPR *fd*. The source and destination are values in format *fmt*.

If GPR *rt* is not zero, then FPR *fs* is not copied and FPR *fd* contains its previous value in format *fm*t. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes **UNPREDICTABLE**.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Floating Point Move Conditional on Zero (cont.) MOVZ.fmt

Operation:

```
if GPR[rt] = 0 then
   StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
   StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation

Multiply and Subtract Word to Hi,Lo **MSUB**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	0 00000	0 00000	MSUB 000100
6	5	5	5	5	6

Format: `MSUB rs, rt` **MIPS32**

Purpose:

To multiply two words and subtract the result from Hi, Lo

Description: `(HI,LO) ← (HI,LO) - (rs × rt)`

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 $HI_{31..0}$ and $LO_{31..0}$.. The most significant 32 bits of the result are sign-extended and written into *HI* and the least signficant 32 bits are sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← ($HI_{31..0}$ || $LO_{31..0}$) - ($GPR[rs]_{31..0}$ × $GPR[rt]_{31..0}$)
HI ← sign_extend($temp_{63..32}$)
LO ← sign_extend($temp_{31..0}$)
```

Exceptions:

None

Programming Notes:

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

Floating Point Multiply Subtract MSUB.fmt

31 26	25 21	20 16	15 11	10 6	5 3	2 0
COP1X 010011	fr	ft	fs	fd	MSUB 101	fmt
6	5	5	5	5	3	3

Format: `MSUB.S fd, fr, fs, ft` **MIPS64**
`MSUB.D fd, fr, fs, ft` **MIPS64**

Purpose:

To perform a combined multiply-then-subtract of FP values

Description: `fd ← (fs × ft) − fr`

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce an intermediate product. The value in FPR *fr* is subtracted from the product. The subtraction result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*.

Cause bits are ORed into the *Flag* bits if no exception is taken.

Restrictions:

The fields *fr, fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, (vfs $\times_{fmt}$ vft) $-_{fmt}$ vfr))
```

Floating Point Multiply Subtract (cont.) MSUB.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

Multiply and Subtract Word to Hi,Lo MSUBU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	0 00000	0 00000	MSUBU 000101
6	5	5	5	5	6

Format: `MSUBU rs, rt` **MIPS32**

Purpose:

To multiply two words and subtract the result from Hi, Lo

Description: `(HI,LO) ← (HI,LO) - (rs × rt)`

The 32-bit word value in GPR *rs* is multiplied by the 32-bit word 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 $HI_{31..0}$ and $LO_{31..0}$.. The most significant 32 bits of the result are sign-extended and written into *HI* and the least signficant 32 bits are sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← $(HI_{31..0} || LO_{31..0}) - ((0^{32} || GPR[rs]_{31..0}) \times (0^{32} || GPR[rt]_{31..0}))$
HI ← sign_extend($temp_{63..32}$)
LO ← sign_extend($temp_{31..0}$)
```

Exceptions:

None

Programming Notes:

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

Move to Coprocessor 0 MTC0

31 26	25 21	20 16	15 11	10 3	2 0
COP0 010000	MT 00100	rt	rd	0 0000 000	sel
6	5	5	5	8	3

Format: `MTC0 rt, rd` **MIPS32**
`MTC0 rt, rd, sel` **MIPS32**

Purpose:

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

Description: `CPR[r0, rd, sel] ← rt`

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

Restrictions:

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

Operation:

```
if (Width(CPR[0,rd,sel]) = 64) then
   CPR[0,rd,sel] ← data
else
   CPR[0,rd,sel] ← $data_{31..0}$
endif
```

Exceptions:

Coprocessor Unusable

Reserved Instruction

Move Word to Floating Point MTC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	MT 00100	rt	fs	0 000 0000 0000
6	5	5	5	11

Format: `MTC1 rt, fs` **MIPS32**

Purpose:

To copy a word from a GPR to an FPU (CP1) general register

Description: `fs ← rt`

The low word in GPR *rt* is placed into the low word of floating point (Coprocessor 1) general register *fs*. If Coprocessor 1 general registers are 64 bits wide, bits 63..32 of register *fs* become undefined.

Restrictions:

None

Operation:

```
data ← GPR[rt]$_{31..0}$
StoreFPR(fs, UNINTERPRETED_WORD, data)
```

Exceptions:

Coprocessor Unusable

Move Word to Coprocessor 2 MTC2

31 26	25 21	20 16	15 11	10	0
COP2 010010	MT 00100	rt	rd	0 000 0000 0	sel
6	5	5	5	8	3

Format: `MTC2 rt, rd` **MIPS32**
`MTC2 rt, rd, sel` **MIPS32**

Purpose:

To copy a word from a GPR to a COP2 general register

Description: `CPR[2,rd,sel] ← rt`

The low word in GPR *rt* is placed into the low word of coprocessor 2 general register specified by the *rd* and *sel* fields. If coprocessor 2 general registers are 64 bits wide, bits 63..32 of register *rd* become undefined. Note that not all coprocessor 2 registers may support the *sel* field. In those instances, the *sel* field must be zero.

Restrictions:

The results are **UNPREDICTABLE** if coprocessor 2 does not contain a register as specified by *rd* and *sel*.

Operation:

```
data ← GPR[rt]$_{31..0}$
CPR[2,rd,sel] ← data
```

Exceptions:

Coprocessor Unusable

Move to HI Register MTHI

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000 0000 0000 0000	MTHI 010001
6	5	15	6

Format: `MTHI rs` **MIPS32**

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:

A computed result written to the *HI*/*LO* pair by DIV, DIVU, DDIV, DDIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either *HI* or *LO*.

Operation:

```
    HI ← GPR[rs]
```

Exceptions:

None

Move to LO Register MTLO

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000 0000 0000 0000	MTLO 010011
6	5	15	6

Format: `MTLO rs` **MIPS32**

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:

A computed result written to the *HI*/*LO* pair by DIV, DIVU, DDIV, DDIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either *HI* or *LO*.

Operation:

```
    LO ← GPR[rs]
```

Exceptions:

None

Multiply Word to GPR MUL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	rd	0 00000	MUL 000010
6	5	5	5	5	6

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

Purpose:

To multiply two words and write the result to a GPR.

Description: `rd ← rs × rt`

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 sign-extended and written to GPR *rd*. The contents of *HI* and *LO* are **UNPREDICTABLE** after the operation. No arithmetic exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Note that this instruction does not provide the capability of writing the result to the HI and LO registers.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
   UndefinedResult()
endif
temp <- GPR[rs] * GPR[rt]
GPR[rd] <- sign_extend($temp_{31..0}$)
HI <- UNPREDICTABLE
LO <- UNPREDICTABLE
```

Exceptions:

None

Programming Notes:

Programs that require overflow detection must check for it explicitly.

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

Floating Point Multiply **MUL.fmt**

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format: `MUL.S fd, fs, ft` **MIPS32**
`MUL.D fd, fs, ft` **MIPS32**

Purpose:

To multiply FP values

Description: `fd ← fs × ft`

The value in FPR *fs* is multiplied by the value in FPR *ft*. The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*.

Restrictions:

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

Operation:

```
    StoreFPR (fd, fmt, ValueFPR(fs, fmt) $\times_{fmt}$ ValueFPR(ft, fmt))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

Multiply Word MULT

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	MULT 011000
6	5	5	10	6

Format: `MULT rs, rt` **MIPS32**

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 sign-extended and placed into special register *LO*, and the high-order 32-bit word is sign-extended and placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
   UndefinedResult()
endif
   prod  ← $GPR[rs]_{31..0} \times GPR[rt]_{31..0}$
   LO    ← sign_extend($prod_{31..0}$)
   HI    ← sign_extend($prod_{63..32}$)
```

Exceptions:

None

Programming Notes:

Programs that require overflow detection must check for it explicitly.

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

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` **MIPS32**

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 sign-extended and placed into special register *LO*, and the high-order 32-bit word is sign-extended and placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UndefinedResult()
endif
   prod← (0 || GPR[rs]$_{31..0}$) × (0 || GPR[rt]$_{31..0}$)
   LO ← sign_extend(prod$_{31..0}$)
   HI ← sign_extend(prod$_{63..32}$)
```

Exceptions:

None

Programming Notes:

Programs that require overflow detection must check for it explicitly.

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

Floating Point Negate NEG.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	NEG 000111
6	5	5	5	5	6

Format: `NEG.S fd, fs` **MIPS32**
`NEG.D fd, fs` **MIPS32**

Purpose:

To negate an FP value

Description: `fd ← −fs`

The value in FPR *fs* is negated and placed into FPR *fd*. The value is negated by changing the sign bit value. The operand and result are values in format *fmt*. This operation is arithmetic; a NaN operand signals invalid operation.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**. The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR(fd, fmt, Negate(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

Floating Point Negative Multiply Add — NMADD.fmt

31 26	25 21	20 16	15 11	10 6	5 3	2 0
COP1X 010011	fr	ft	fs	fd	NMADD 110	fmt
6	5	5	5	5	3	3

Format: `NMADD.S fd, fr, fs, ft` **MIPS64**
`NMADD.D fd, fr, fs, ft` **MIPS64**

Purpose:

To negate a combined multiply-then-add of FP values

Description: `fd ← − ((fs × ft) + fr)`

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce an intermediate product. The value in FPR *fr* is added to the product.

The result sum is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, negated by changing the sign bit, and placed into FPR *fd*. The operands and result are values in format *fmt*.

Cause bits are ORed into the *Flag* bits if no exception is taken.

Restrictions:

The fields *fr*, *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, −(vfr $+_{fmt}$ (vfs $\times_{fmt}$ vft)))
```

Floating Point Negative Multiply Add (cont.) NMADD.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

Floating Point Negative Multiply Subtract **NMSUB.fmt**

31 26	25 21	20 16	15 11	10 6	5 3	2 0
COP1X 010011	fr	ft	fs	fd	NMSUB 111	fmt
6	5	5	5	5	3	3

Format: `NMSUB.S fd, fr, fs, ft` **MIPS64**
`NMSUB.D fd, fr, fs, ft` **MIPS64**

Purpose:

To negate a combined multiply-then-subtract of FP values

Description: `fd ← - ((fs × ft) - fr)`

The value in FPR *fs* is multiplied by the value in FPR *ft* to produce an intermediate product. The value in FPR *fr* is subtracted from the product.

The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, negated by changing the sign bit, and placed into FPR *fd*. The operands and result are values in format *fmt*.

Cause bits are ORed into the *Flag* bits if no exception is taken.

Restrictions:

The fields *fr, fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, −((vfs $\times_{fmt}$ vft) $-_{fmt}$ vfr))
```

Floating Point Negative Multiply Subtract (cont.) NMSUB.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow

Not Or **NOR**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	NOR 100111
6	5	5	5	5	6

Format: `NOR rd, rs, rt` **MIPS32**

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 000000	rs	rt	rd	0 00000	OR 100101
6	5	5	5	5	6

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

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` **MIPS32**

Purpose:

To do a bitwise logical OR with a constant

Description: `rt ← 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 26	25 21	20 16	15 0
PREF 110011	base	hint	offset
6	5	5	16

Format: `PREF hint,offset(base)` **MIPS32**

Purpose:

To move data between memory and cache.

Description: `prefetch_memory(base+offset)`

PREF adds the 16-bit signed *offset* to the contents of GPR *base* to form an effective byte address. 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 movement occurs.However even if no data is prefetched, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

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.

Prefetch (cont.) PREF

Any of the following conditions causes the core to treat a PREF instruction as a NOP.

- A reserved *hint* value is used
- The address has a translation error
- The address maps to an uncacheable page
- The cache controller is not idle as there are outstanding transactions, i.e. refills, loads, stores, etc.

In all other cases, except when *hint* equals 25, 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.

Prefetch (cont.) PREF

Table 12-28 Values of the *hint* Field for the PREF Instruction

Value	Name	Data Use and Desired Prefetch Action
0	load	Use: Prefetched data is expected to be read (not modified). Action: Fetch data as if for a load.
1	store	Use: Prefetched data is expected to be stored or modified. Action: Fetch data as if for a store.
2-3	Reserved	Reserved - treated as a NOP.
4	load_streamed	Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache. Action: Fetch data as if for a load.
5	store_streamed	Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache. Action: Fetch data as if for a store.
6	load_retained	Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a load.
7	store_retained	Use: Prefetched data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a store.
8-24	Reserved	Reserved - treated as a NOP.

Table 12-28 Values of the *hint* Field for the PREF Instruction

25	writeback_invalidate (also known as “nudge”)	Action: Schedule a writeback of any dirty data. The cache line is marked as invalid upon completion of the writeback or if the line was found clean.
26-29	Reserved	Reserved - treated as a NOP.
30	Reserved	Reserved - treated as a NOP.
31	Reserved	Reserved - treated as a NOP.

Prefetch (cont.) PREF

Restrictions:

None

Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

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

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.

Prefetch Indexed PREFX

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	hint	0 00000	PREFX 001111
6	5	5	5	5	6

Format: `PREFX hint, index(base)` **MIPS64**

Purpose:

To move data between memory and cache.

Description: `prefetch_memory[base+index]`

PREFX adds the contents of GPR *index* to the contents of GPR *base* to form an effective byte address. The *hint* field supplies information about the way the data is expected to be used.

The only functional difference between the PREF and PREFX instructions is the addressing mode implemented by the two. Refer to the PREF instruction for all other details, including the encoding of the *hint* field.

Restrictions:

None

Operation:

```
vAddr ← GPR[base] + GPR[index]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Programming Notes:

Refer to the corresponding section in the PREF instruction description.

Reciprocal Approximation RECIP.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	RECIP 010101
6	5	5	5	5	6

Format: `RECIP.S fd, fs` **MIPS64**
`RECIP.D fd, fs` **MIPS64**

Purpose:

To approximate the reciprocal of an FP value (quickly)

Description: `fd ← 1.0 / fs`

The reciprocal of the value in FPR *fs* is approximated and placed into FPR *fd*. The operand and result are values in format *fmt*.

The numeric accuracy of this operation does not meet the accuracy specified by the IEEE 754 Floating Point standard. The computed result differs from the both the exact result and the IEEE-mandated representation of the exact result by no more than one unit in the least-significant place (ULP).

The computed result is not affected by the current rounding mode in FCSR unless an underflow occurs in which case the default result (Zero or MinNorm) is determined by the current rounding mode in FCSR.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of RECIP.D is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
StoreFPR(fd, fmt, 1.0 / valueFPR(fs, fmt))
```

Reciprocal Approximation (cont.) RECIP.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Division-by-zero, Unimplemented Op, Invalid Op, Overflow, Underflow

Floating Point Round to Long Fixed Point ROUND.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ROUND.L 001000
6	5	5	5	5	6

Format: `ROUND.L.S   fd, fs` **MIPS64**
`ROUND.L.D   fd, fs` **MIPS64**

Purpose:

To convert an FP value to 64-bit fixed point, rounding to nearest

Description: `fd ← convert_and_round(fs)`

The value in FPR *fs*, in format *fmt*, is converted to a value in 64-bit long fixed point format and rounded to nearest/even (rounding mode 0). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}–1$, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Floating Point Round to Long Fixed Point (cont.) ROUND.L.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow

Floating Point Round to Word Fixed Point ROUND.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ROUND.W 001100
6	5	5	5	5	6

Format: `ROUND.W.S   fd, fs` **MIPS32**
`ROUND.W.D   fd, fs` **MIPS32**

Purpose:

To convert an FP value to 32-bit fixed point, rounding to nearest

Description: `fd ← convert_and_round(fs)`

The value in FPR *fs,* in format *fmt*, is converted to a value in 32-bit word fixed point format rounding to nearest/even (rounding mode 0). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}–1$, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Floating Point Round to Word Fixed Point (cont). ROUND.W.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow

Reciprocal Square Root Approximation RSQRT.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	RSQRT 010110
6	5	5	5	5	6

Format: `RSQRT.S fd, fs` **MIPS64**
`RSQRT.D fd, fs` **MIPS64**

Purpose:

To approximate the reciprocal of the square root of an FP value (quickly)

Description: `fd ← 1.0 / sqrt(fs)`

The reciprocal of the positive square root of the value in FPR *fs* is approximated and placed into FPR *fd*. The operand and result are values in format *fmt*.

The numeric accuracy of this operation does not meet the accuracy specified by the IEEE 754 Floating Point standard. The computed result differs from both the exact result and the IEEE-mandated representation of the exact result by no more than two units in the least-significant place (ULP).

The computed result is not affected by the current rounding mode in FCSR.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of RSQRT.D is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
StoreFPR(fd, fmt, 1.0 / SquareRoot(valueFPR(fs, fmt)))
```

Reciprocal Square Root Approximation (cont.) RSQRT.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Division-by-zero, Unimplemented Operation, Invalid Operation, Overflow, Underflow

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)` **MIPS32**

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       ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor $ReverseEndian^3$)
bytesel     ← $vAddr_{2..0}$ xor $BigEndianCPU^3$
datadoubleword← $GPR[rt]_{63-8*bytesel..0}$ || $0^{8*bytesel}$
StoreMemory (CCA, BYTE, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Store Conditional Word SC

31 26	25 21	20 16	15 0
SC 111000	base	rt	offset
6	5	5	16

Format: `SC rt, offset(base)` **MIPS32**

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:

- An ERET instruction is executed.

If either of the following events occurs between the execution of LL and SC, the SC may succeed or it may fail; the success or failure is not predictable. Portable programs should not cause one of these events.

- A load, store, or prefetch is executed on the processor executing the LL/SC.
- The instructions executed starting with the LL and ending with the SC do not lie in a 2048-byte contiguous region of virtual memory. The region does not have to be aligned, other than the alignment required for instruction words.

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.

Store Conditional Word (cont.) SC

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 $vAddr_{1..0} \neq 0^2$ then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || $0^2$))
bytesel← $vAddr_{2..0}$ xor (BigEndianCPU || $0^2$)
datadoubleword← $GPR[rt]_{63-8*bytesel..0}$ || $0^{8*bytesel}$
if LLbit then
   StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
endif
GPR[rt]← $0^{63}$ || LLbit
```

Store Conditional Word (cont.) SC

Exceptions:

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

Programming Notes:

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

```
L1:
    LL     T1, (T0)  # load counter
    ADDI   T2, T1, 1 # increment
    SC     T2, (T0)  # try to store, checking for atomicity
    BEQ    T2, 0, L1 # if not atomic (0), try again
    NOP              # branch-delay slot
```

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

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

Store Conditional Doubleword SCD

31 26	25 21	20 16	15 0
SCD 111100	base	rt	offset
6	5	5	16

Format: `SCD rt, offset(base)` **MIPS64**

Purpose:

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

Description:`if atomic_update then memory[base+offset] ← rt, rt ← 1 else rt ← 0`

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The SCD completes the RMW sequence begun by the preceding LLD instruction executed on the processor.

If it would complete the RMW sequence atomically, the following occur**:**

- The 64-bit doubleword 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 LLD and SCD, the SCD fails:

- An ERET instruction is executed.

If either of the following events occurs between the execution of LLD and SCD, the SCD may succeed or it may fail; success or failure is not predictable. Portable programs should not cause these events:

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LLD/SCD.
- The instructions executed starting with the LLD and ending with the SCD do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

The following two conditions must be true or the result of the SCD is undefined:

- Execution of the SCD must be preceded by execution of an LLD instruction.
- An RMW sequence executed without intervening exceptions must use the same address in the LLD and SCD. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.

Store Conditional Doubleword (cont.) SCD

Restrictions:

The 64-bit doubleword of register *rt* is conditionally 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.

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

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if $vAddr_{2..0} \neq 0^3$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
datadoubleword ← GPR[rt]
if LLbit then
   StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
endif
GPR[rt] ← $0^{63}$ || LLbit
```

Store Conditional Doubleword (cont.) SCD

Exceptions:

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

Programming Notes:

LLD and SCD are used to atomically update memory locations, as shown below.

```
L1:
    LLD    T1, (T0)  # load counter
    ADDI   T2, T1, 1 # increment
    SCD    T2, (T0)  # try to store,
                     #  checking for atomicity
    BEQ    T2, 0, L1 # if not atomic (0), try again
    NOP              # branch-delay slot
```

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

LLD and SCD 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.

Store Doubleword SD

31 26	25 21	20 16	15 0
SD 111111	base	rt	offset
6	5	5	16

Format: `SD rt, offset(base)` **MIPS64**

Purpose:

To store a doubleword to memory

Description: `memory[base+offset] ← rt`

The 64-bit doubleword in GPR *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 any of the 3 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr$_{2..0}$ ≠ $0^3$ then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
datadoubleword← GPR[rt]
StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Software Debug Breakpoint SDBBP

31 26	25 6	5 0
SPECIAL2 011100	code	SDBBP 111111
6	20	6

Format: SDBBP code **EJTAG**

Purpose:

To cause a debug breakpoint exception

Description:

This instruction causes a debug exception, passing control to the debug exception handler. The code field can be used for passing information to the debug exception handler, and is retrieved by the debug exception handler only by loading the contents of the memory word containing the instruction, using the DEPC register. The CODE field is not used in any way by the hardware.

Restrictions:

None

Operation:

```
If $Debug_{DM}$ = 0 then
   SignalDebugBreakpointException()
else
   SignalDebugModeBreakpointException()
endif
```

Exceptions:

Debug Breakpoint Exception

Store Doubleword from Floating Point SDC1

31 26	25 21	20 16	15 0
SDC1 111101	base	ft	offset
6	5	5	16

Format: `SDC1 ft, offset(base)` **MIPS32**

Purpose:

To store a doubleword from an FPR to memory

Description: `memory[base+offset] ← ft`

The 64-bit doubleword in FPR *ft* 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:

An Address Error exception occurs if EffectiveAddress$_{2..0}$ ≠ 0 (not doubleword-aligned).

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr$_{2..0}$ ≠ $0^3$ then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error

Store Doubleword from Coprocessor 2 SDC2

31 26	25 21	20 16	15 0
SDC2 111110	base	rt	offset
6	5	5	16

Format: `SDC2 rt, offset(base)` **MIPS32**

Purpose:

To store a doubleword from a Coprocessor 2 register to memory

Description: `memory[base+offset] ← rt`

The 64-bit doubleword in Coprocessor 2 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:

An Address Error exception occurs if EffectiveAddress$_{2..0}$ ≠ 0 (not doubleword-aligned).

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if $vAddr_{2..0} \ne 0^3$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← CPR[2,rt,0]
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error

Store Doubleword Left — SDL

31 — 26	25 — 21	20 — 16	15 — 0
SDL 101100	base	rt	offset
6	5	5	16

Format: `SDL rt, offset(base)` **MIPS64**

Purpose:

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

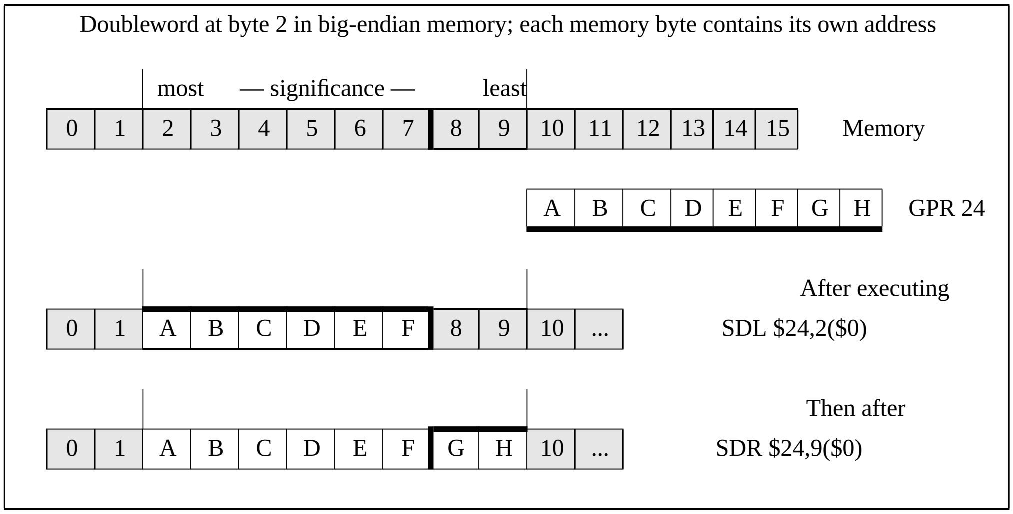
Description: `memory[base+offset] ← Some_Bytes_From rt`

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 8 consecutive bytes forming a doubleword *(DW*) in memory, starting at an arbitrary byte boundary.

A part of *DW*, the most-significant 1 to 8 bytes, is in the aligned doubleword containing *EffAddr.* The same number of most-significant (left) bytes of GPR *rt* are stored into these bytes of *DW.*

The figure below illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 6 bytes, is located in the aligned doubleword containing the most-significant byte at 2. First, SDL stores the 6 most-significant bytes of the source register into these bytes in memory. Next, the complementary SDR instruction stores the remainder of *DW.*

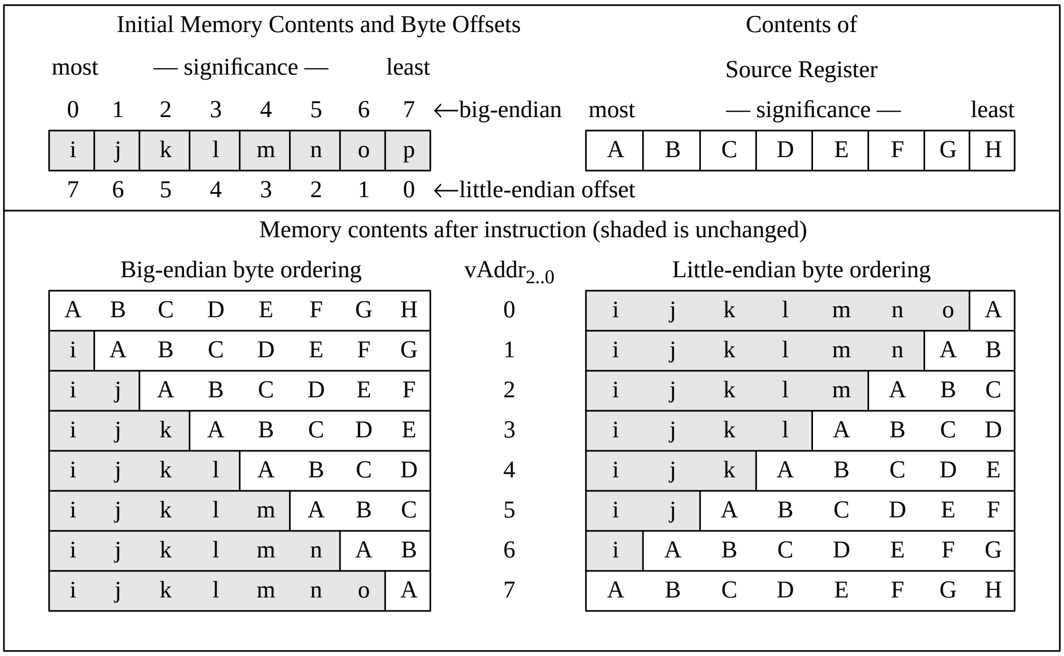
Figure 12-13 Unaligned Doubleword Store With SDL and SDR


Doubleword at byte 2 in big-endian memory; each memory byte contains its own address
most — significance — least
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Memory
A B C D E F G H
GPR 24
After executing
0 1 A B C D E F 8 9 10 ...
SDL $24,2($0)
Then after
0 1 A B C D E F G H 10 ...
SDR $24,9($0)

Store Doubleword Left (cont.) SDL

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned doubleword—that is, the low 3 bits of the address (*vAddr2..0*)—and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes stored for every combination of offset and byte ordering.

Figure 12-14 Bytes Stored by an SDL Instruction



Initial Memory Contents and Byte Offsets

most — significance — least

0	1	2	3	4	5	6	7	←big-endian
i	j	k	l	m	n	o	p	
7	6	5	4	3	2	1	0	←little-endian offset

Contents of Source Register

most — significance — least

A	B	C	D	E	F	G	H

Memory contents after instruction (shaded is unchanged)

Big-endian byte ordering								$vAddr_{2..0}$	Little-endian byte ordering							
A	B	C	D	E	F	G	H	0	i	j	k	l	m	n	o	A
i	A	B	C	D	E	F	G	1	i	j	k	l	m	n	A	B
i	j	A	B	C	D	E	F	2	i	j	k	l	m	A	B	C
i	j	k	A	B	C	D	E	3	i	j	k	l	A	B	C	D
i	j	k	l	A	B	C	D	4	i	j	k	A	B	C	D	E
i	j	k	l	m	A	B	C	5	i	j	A	B	C	D	E	F
i	j	k	l	m	n	A	B	6	i	A	B	C	D	E	F	G
i	j	k	l	m	n	o	A	7	A	B	C	D	E	F	G	H

Restrictions:

None

Store Doubleword Left (cont.) SDL

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ←  pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor ReverseEndian$^{3}$)
If BigEndianMem = 0 then
   pAddr← pAddr$_{PSIZE-1..3}$ || 0$^{3}$
endif
bytesel← vAddr$_{2..0}$ xor BigEndianCPU$^{3}$
datadoubleword← 0$^{56-8*bytesel}$ || GPR[rt]$_{63..56-8*bytesel}$
StoreMemory (CCA, byte, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Store Doubleword Right SDR

31 26	25 21	20 16	15 0
SDR 101101	base	rt	offset
6	5	5	16

Format: `SDR rt, offset(base)` **MIPS64**

Purpose:

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

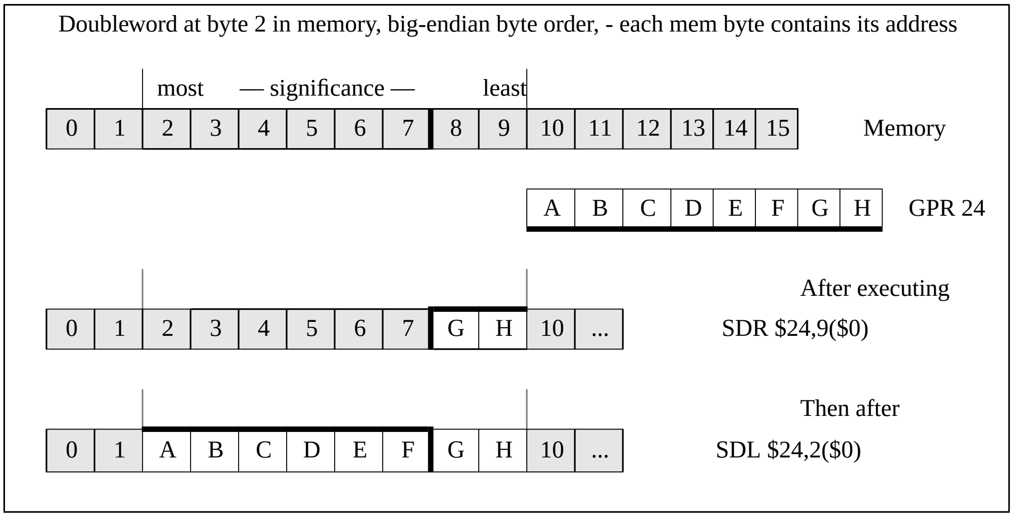
Description: `memory[base+offset] ← Some_Bytes_From rt`

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 8 consecutive bytes forming a doubleword *(DW*) in memory, starting at an arbitrary byte boundary.

A part of *DW*, the least-significant 1 to 8 bytes, is in the aligned doubleword containing *EffAddr.* The same number of least-significant (right) bytes of GPR *rt* are stored into these bytes of *DW.*

The figure below illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of *DW*, 2 bytes, is located in the aligned doubleword containing the least-significant byte at 9. First, SDR stores the 2 least-significant bytes of the source register into these bytes in memory. Next, the complementary SDL stores the remainder of *DW.*

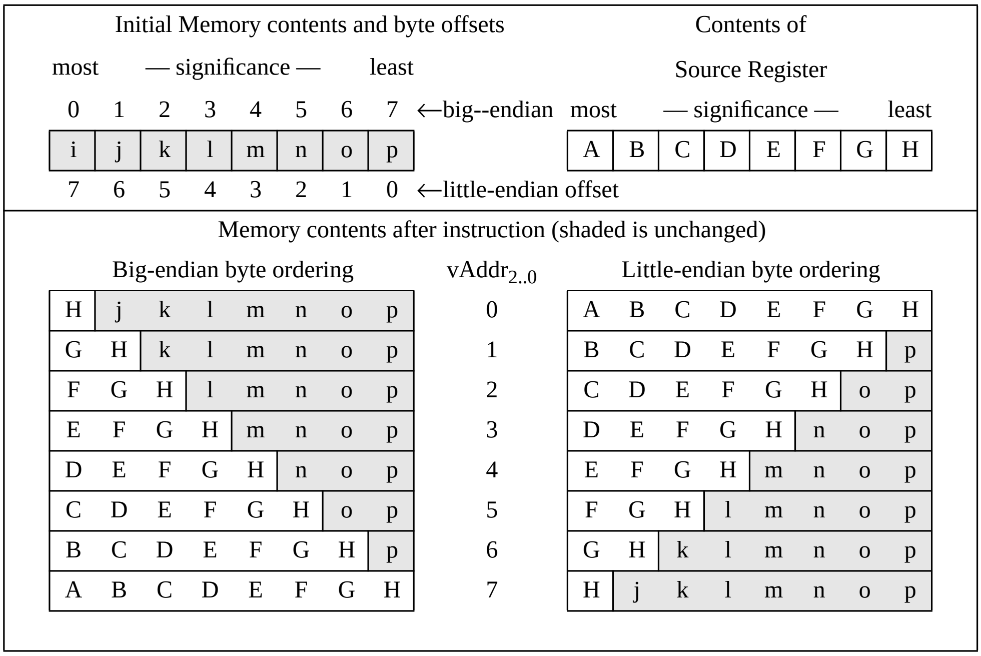
Figure 12-15 Unaligned Doubleword Store With SDR and SDL


Doubleword 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 9 10 11 12 13 14 15
Memory
A B C D E F G H
GPR 24
After executing
0 1 2 3 4 5 6 7 G H 10 ...
SDR $24,9($0)
Then after
0 1 A B C D E F G H 10 ...
SDL $24,2($0)

Store Doubleword Right (cont.) SDR

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned doubleword—that is, the low 3 bits of the address (*vAddr2..0*)—and the current byte ordering mode of the processor (big- or little-endian). Figure 12-16 shows the bytes stored for every combination of offset and byte-ordering.

Figure 12-16 Bytes Stored by an SDR Instruction



Initial Memory contents and byte offsets

most — significance — least

0	1	2	3	4	5	6	7	←big--endian
i	j	k	l	m	n	o	p	
7	6	5	4	3	2	1	0	←little-endian offset

Contents of Source Register

most — significance — least

A	B	C	D	E	F	G	H

Memory contents after instruction (shaded is unchanged)

Big-endian byte ordering								$vAddr_{2..0}$	Little-endian byte ordering							
H	j	k	l	m	n	o	p	0	A	B	C	D	E	F	G	H
G	H	k	l	m	n	o	p	1	B	C	D	E	F	G	H	p
F	G	H	l	m	n	o	p	2	C	D	E	F	G	H	o	p
E	F	G	H	m	n	o	p	3	D	E	F	G	H	n	o	p
D	E	F	G	H	n	o	p	4	E	F	G	H	m	n	o	p
C	D	E	F	G	H	o	p	5	F	G	H	l	m	n	o	p
B	C	D	E	F	G	H	p	6	G	H	k	l	m	n	o	p
A	B	C	D	E	F	G	H	7	H	j	k	l	m	n	o	p

Restrictions:

None

Store Doubleword Right (cont.) SDR

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ←  pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$  xor  ReverseEndian$^3$)
If BigEndianMem = 0 then
   pAddr← pAddr$_{PSIZE-1..3}$ || 0$^3$
endif
bytesel← vAddr$_{1..0}$ xor BigEndianCPU$^3$
datadoubleword← GPR[rt]$_{63-8*bytesel}$ || 0$^{8*bytesel}$
StoreMemory (CCA, DOUBLEWORD-byte, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Store Doubleword Indexed from Floating Point SDXC1

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	fs	0 00000	SDXC1 001001
6	5	5	5	5	6

Format: `SDXC1 fs, index(base)` **MIPS64**

Purpose:

To store a doubleword from an FPR to memory (GPR+GPR addressing)

Description: `memory[base+index] ← fs`

The 64-bit doubleword in FPR *fs* is stored in memory at the location specified by the aligned effective address. The contents of GPR *index* and GPR *base* are added to form the effective address.

Restrictions:

An Address Error exception occurs if $\text{EffectiveAddress}_{2..0} \neq 0$ (not doubleword-aligned).

Operation:

```
vAddr ← GPR[base] + GPR[index]
if $vAddr_{2..0} \neq 0^3$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Coprocessor Unusable, Address Error, Reserved Instruction.

Store Halfword SH

31 26	25 21	20 16	15 0
SH 101001	base	rt	offset
6	5	5	16

Format: `SH rt, offset(base)` **MIPS32**

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 vAddr$_0$ ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr$_{PSIZE-1..3}$ || (pAddr1$_{2..0}$ xor (ReverseEndian$^2$ || 0))
bytesel← vAddr1$_{2..0}$ xor (BigEndianCPU$^2$ || 0)
datadoubleword← GPR[rt]$_{63-8*bytesel..0}$ || 0$^{8*bytesel}$
StoreMemory (CCA, HALFWORD, datadoubleword, 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` **MIPS32**

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 sign-extended and 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:

Unlike nearly all other word operations, the SLL input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign-extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign-extends it.

SLL r0, r0, 0, expressed as NOP, is the assembly idiom used to denote no operation.

SLL r0, r0, 1, expressed as SSNOP, is the assembly idiom used to denote no operation that causes an issue break on superscalar processors.

Shift Word Left Logical Variable **SLLV**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLLV 000100
6	5	5	5	5	6

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

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 sign-extended and placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions: None

Operation:

```
s       ← GPR[rs]$_{4..0}$
temp    ← GPR[rt]$_{(31-s)..0}$ || 0$^{s}$
GPR[rd]← sign_extend(temp)
```

Exceptions: None

Programming Notes:

Unlike nearly all other word operations, the input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign-extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign-extends it.

Set on Less Than SLT

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLT 101010
6	5	5	5	5	6

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

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` **MIPS32**

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 Unsigned SLTIU

31 26	25 21	20 16	15 0
SLTIU 001011	rs	rt	immediate
6	5	5	16

Format: `SLTIU rt, rs, immediate` **MIPS32**

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 000000	rs	rt	rd	0 00000	SLTU 101011
6	5	5	5	5	6

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

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] ← $0^{GPRLEN-1}$ || 1
else
    GPR[rd] ← $0^{GPRLEN}$
endif
```

Exceptions:

None

Floating Point Square Root SQRT.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	SQRT 000100
6	5	5	5	5	6

Format: `SQRT.S fd, fs` **MIPS32**
`SQRT.D fd, fs` **MIPS32**

Purpose:

To compute the square root of an FP value

Description: `fd ← SQRT(fs)`

The square root of the value in FPR *fs* is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operand and result are values in format *fmt*.

If the value in FPR *fs* corresponds to – 0, the result is – 0.

Restrictions:

If the value in FPR *fs* is less than 0, an Invalid Operation condition is raised.

The fields *fs* and *fd* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR(fd, fmt, SquareRoot(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Inexact, Unimplemented Operation

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` **MIPS32**

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 sign-extended and placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if NotWordValue(GPR[rt]) then
   UndefinedResult()
endif
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` **MIPS32**

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 sign-extended and placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if NotWordValue(GPR[rt]) then
   UndefinedResult()
endif
s      ← $GPR[rs]_{4..0}$
temp   ← $(GPR[rt]_{31})^{s}$ || $GPR[rt]_{31..s}$
GPR[rd]← 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` **MIPS32**

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 sign-extended and placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if NotWordValue(GPR[rt]) then
   UndefinedResult()
endif
s      ← sa
temp   ← $0^s$ || $\text{GPR[rt]}_{31..s}$
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 000000	rs	rt	rd	0 00000	SRLV 000110
6	5	5	5	5	6

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

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 sign-extended and placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if NotWordValue(GPR[rt]) then
   UndefinedResult()
endif
s       ← $\text{GPR[rs]}_{4..0}$
temp    ← $0^{s}$ || $\text{GPR[rt]}_{31..s}$
GPR[rd]← sign_extend(temp)
```

Exceptions:

None

Superscalar No Operation SSNOP

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	0 00000	0 00000	1 00001	SLL 000000
6	5	5	5	5	6

Format: `SSNOP` **MIPS32**

Purpose:

Break superscalar issue on a superscalar processor.

Description:

SSNOP is the assembly idiom used to denote superscalar no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 1.

This instruction alters the instruction issue behavior on a superscalar processor by forcing the SSNOP instruction to single-issue. The processor must then end the current instruction issue between the instruction previous to the SSNOP and the SSNOP. The SSNOP then issues alone in the next issue slot.

On a single-issue processor, this instruction is a NOP that takes an issue slot.

Restrictions:

None

Operation:

None

Exceptions:

None

Programming Notes:

SSNOP is intended for use primarily to allow the programmer control over CP0 hazards by converting instructions into cycles in a superscalar processor. For example, to insert at least two cycles between an MTC0 and an ERET, one would use the following sequence:

```
mtc0   x,y
ssnop
ssnop
eret
```

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` **MIPS32**

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 sign-extended and placed into GPR *rd*.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UndefinedResult()
endif
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:

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

Floating Point Subtract SUB.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	SUB 000001
6	5	5	5	5	6

Format: `SUB.S fd, fs, ft` **MIPS32**
`SUB.D fd, fs, ft` **MIPS32**

Purpose:

To subtract FP values

Description: `fd ← fs - ft`

The value in FPR *ft* is subtracted from the value in FPR *fs*. The result is calculated to infinite precision, rounded according to the current rounding mode in *FCSR*, and placed into FPR *fd*. The operands and result are values in format *fmt*. **Restrictions:**

The fields *fs*, *ft*, and *fd* must specify FPRs valid for operands of type *fmt*. If they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format *fmt*; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR (fd, fmt, ValueFPR(fs, fmt) $-_{fmt}$ ValueFPR(ft, fmt))
```

CPU Exceptions:

Coprocessor Unusable, Reserved Instruction

FPU Exceptions:

Inexact, Overflow, Underflow, Invalid Op, Unimplemented Op

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` **MIPS32**

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 sign-extended and placed into GPR *rd*.

No integer overflow exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UndefinedResult()
endif
temp   ← GPR[rs] - GPR[rt]
GPR[rd]← sign_extend(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 Doubleword Indexed Unaligned from Floating Point SUXC1

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	fs	0 00000	SUXC1 001101
6	5	5	5	5	6

Format: `SUXC1 fs, index(base)` **MIPS64**

Purpose:

To store a doubleword from an FPR to memory (GPR+GPR addressing) ignoring alignment

Description: memory[(base+index)$_{PSIZE-1..3}$] ← fs

The contents of the 64-bit doubleword in FPR *fs* is stored at the memory location specified by the effective address. The contents of GPR *index* and GPR *base* are added to form the effective address. The effective address is doubleword-aligned; EffectiveAddress$_{2..0}$ are ignored.

Restrictions:

The result of this instruction is undefined if the processor is executing in 16 FP registers mode.

Operation:

```
vAddr ← (GPR[base]+GPR[index])$_{63..3}$ || 0$^{3}$
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr  ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor (ReverseEndian || 0$^{2}$))
bytesel← vAddr$_{2..0}$ xor (BigEndianCPU || 0$^{2}$)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_WORD) || 0$^{8*bytesel}$
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified

Store Word SW

31 26	25 21	20 16	15 0
SW 101011	base	rt	offset
6	5	5	16

Format: `SW rt, offset(base)` **MIPS32**

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)
pAddr  ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor (ReverseEndian || 0$^{2}$))
bytesel← vAddr$_{2..0}$ xor (BigEndianCPU || 0$^{2}$)
datadoubleword← GPR[rt]$_{63-8*bytesel..0}$ || 0$^{8*bytesel}$
StoreMemory (CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error

Store Word from Floating Point SWC1

31 26	25 21	20 16	15 0
SWC1 111001	base	ft	offset
6	5	5	16

Format: `SWC1 ft, offset(base)` **MIPS32**

Purpose:

To store a word from an FPR to memory

Description: `memory[base+offset] ← ft`

The low 32-bit word from FPR *ft* 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:

An Address Error exception occurs if EffectiveAddress$_{1..0}$ ≠ 0 (not word-aligned).

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr$_{1..0}$ ≠ 0$^{3}$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr  ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor (ReverseEndian || 0$^{2}$))
bytesel← vAddr$_{2..0}$ xor (BigEndianCPU || 0$^{2}$)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_WORD) || 0$^{8*bytesel}$
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error

Store Word from Coprocessor 2 — SWC2

31 — 26	25 — 21	20 — 16	15 — 0
SWC2 111010	base	rt	offset
6	5	5	16

Format: `SWC2 rt, offset(base)` **MIPS32**

Purpose:

To store a word from a COP2 register to memory

Description: `memory[base+offset] ← ft`

The low 32-bit word from COP2 (Coprocessor 2) 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:

An Address Error exception occurs if EffectiveAddress$_{1..0}$ ≠ 0 (not word-aligned).

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if $vAddr_{2..0} \neq 0^3$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || $0^2$))
bytesel ← $vAddr_{2..0}$ xor (BigEndianCPU || $0^2$)
datadoubleword ← $CPR[2,rt,0]_{63-8*bytesel..0}$ || $0^{8*bytesel}$
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error

Store Word Left SWL

31 26	25 21	20 16	15 0
SWL 101010	base	rt	offset
6	5	5	16

Format: `SWL rt, offset(base)` **MIPS32**

Purpose:

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

Description: `memory[base+offset] ← rt`

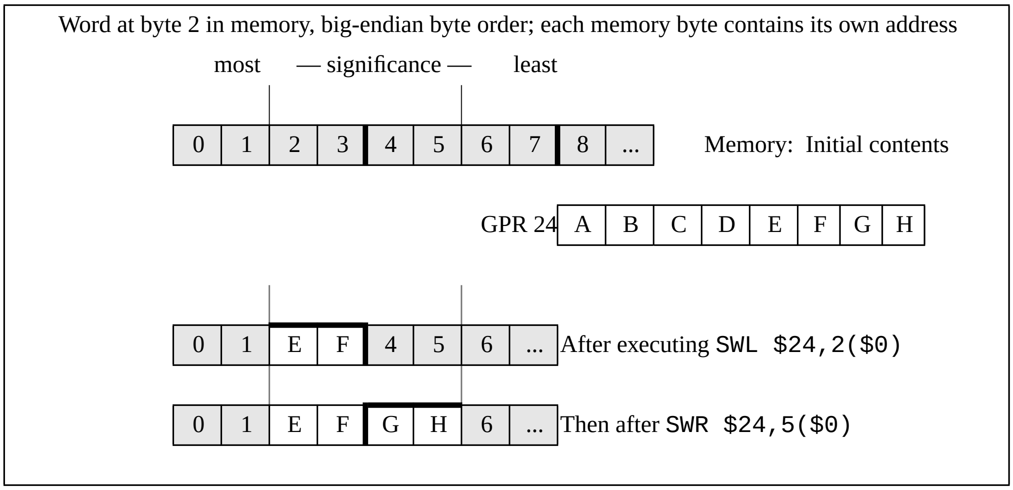
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

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*.

If GPR *rt* is a 64-bit register, the source word is the low word of the register.

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

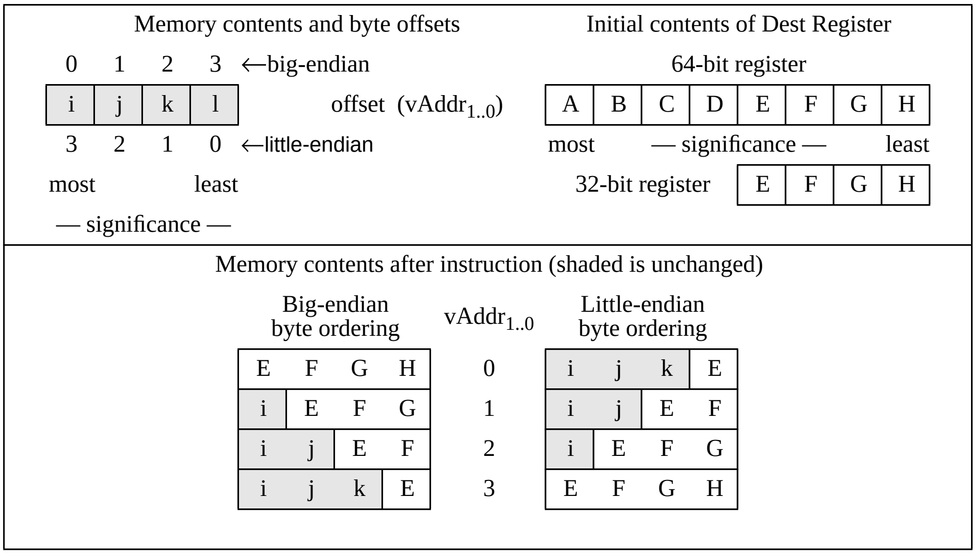
Figure 12-17 Unaligned Word Store Using SWL and SWR


Word at byte 2 in memory, big-endian byte order; each memory byte contains its own address
most — significance — least
0 1 2 3 4 5 6 7 8 ...
Memory: Initial contents
GPR 24
A B C D E F G H
0 1 E F 4 5 6 ...
After executing SWL $24,2($0)
0 1 E F G H 6 ...
Then after SWR $24,5($0)

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). The following figure shows the bytes stored for every combination of offset and byte ordering.

Store Word Left (cont.) SWL

Figure 12-18 Bytes Stored by an SWL Instruction



Big-endian byte ordering	vAddr$_{1..0}$	Little-endian byte ordering
E F G H	0	i j k E
i E F G	1	i j E F
i j E F	2	i E F G
i j k E	3	E F G H

Restrictions:

None

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor  ReverseEndian$^3$)
If BigEndianMem = 0 then
   pAddr ← pAddr$_{PSIZE-1..2}$ || 0$^2$
endif
byte   ← vAddr$_{1..0}$ xor BigEndianCPU$^2$
if (vAddr$_2$ xor BigEndianCPU) = 0 then
   datadoubleword ← 0$^{32}$ || 0$^{24-8*byte}$ || GPR[rt]$_{31..24-8*byte}$
else
   datadoubleword ← 0$^{24-8*byte}$ || GPR[rt]$_{31..24-8*byte}$ || 0$^{32}$
endif

StoreMemory(CCA, byte, datadoubleword, 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 101110	base	rt	offset
6	5	5	16

Format: `SWR rt, offset(base)` **MIPS32**

Purpose:

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

Description: `memory[base+offset] ← rt`

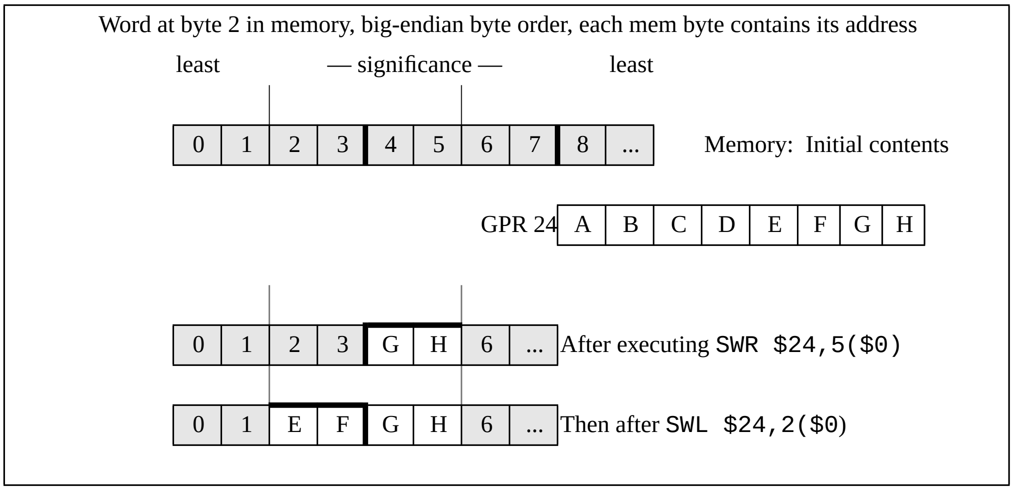
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

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*.

If GPR *rt* is a 64-bit register, the source word is the low word of the register.

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

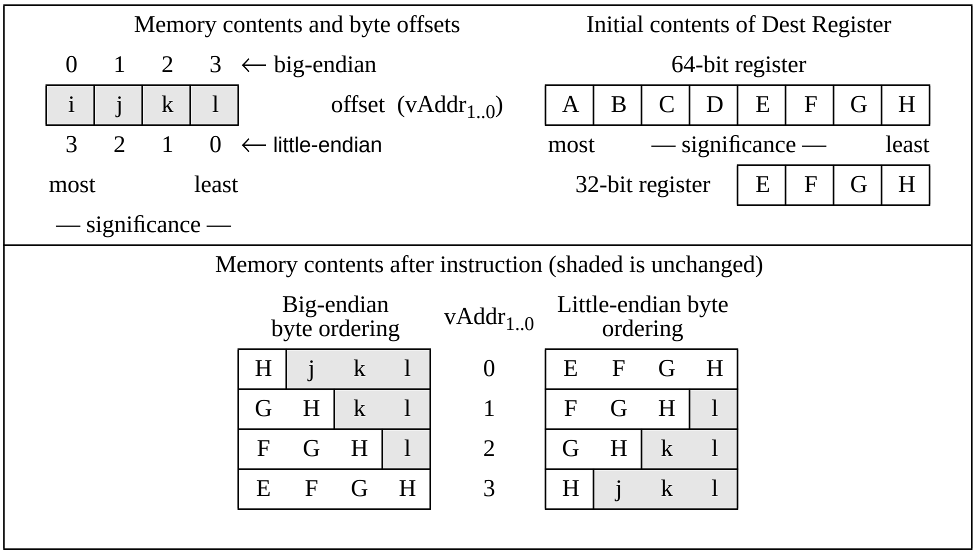
Figure 12-19 Unaligned Word Store Using SWR and SWL



Store Word Right (cont.) 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). The following figure shows the bytes stored for every combination of offset and byte-ordering.

Figure 12-20 Bytes Stored by SWR Instruction



Restrictions:

None

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor $ReverseEndian^3$)
If BigEndianMem = 0 then
   pAddr ← $pAddr_{PSIZE-1..2}$ || $0^2$
endif
byte   ← $vAddr_{1..0}$ xor $BigEndianCPU^2$
if ($vAddr_2$ xor BigEndianCPU) = 0 then
   datadoubleword ← $0^{32}$ || $GPR[rt]_{31-8*byte..0}$ || $0^{8*byte}$
else
   datadoubleword ← $GPR[rt]_{31-8*byte..0}$ || $0^{8*byte}$ || $0^{32}$
endif

StoreMemory(CCA, WORD-byte, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

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

Store Word Indexed from Floating Point SWXC1

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	fs	0 00000	SWXC1 001000
6	5	5	5	5	6

Format: `SWXC1 fs, index(base)` **MIPS64**

Purpose:

To store a word from an FPR to memory (GPR+GPR addressing)

Description: `memory[base+index] ← fs`

The low 32-bit word from FPR *fs* is stored in memory at the location specified by the aligned effective address. The contents of GPR *index* and GPR *base* are added to form the effective address.

Restrictions:

An Address Error exception occurs if $\text{EffectiveAddress}_{1..0} \neq 0$ (not word-aligned).

Operation:

```
vAddr ← GPR[base] + GPR[index]
if $vAddr_{1..0} \neq 0^3$ then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr  ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || $0^2$))
bytesel← $vAddr_{2..0}$ xor (BigEndianCPU || $0^2$)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_WORD) || $0^{8*bytesel}$
StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Reserved Instruction, Coprocessor Unusable

Synchronize Shared Memory SYNC

31 26	25 21 20 16 15 11	10 6	5 0
SPECIAL 000000	0 00 0000 0000 0000 0	stype	SYNC 001111
6	15	5	6

Format: `SYNC (stype = 0 implied)` **MIPS32**

Purpose:

To order loads and stores.

Description:

Simple Description:

- SYNC 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 is required, potentially in conjunction with SSNOP, to guarantee that memory reference results are visible across operating mode changes. For example, a SYNC is required on entry to and exit from Debug Mode to guarantee that memory affects are handled correctly.

Detailed Description:

- 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.
- The SYNC instruction stalls until all loads, stores, refills are completed and all write buffers are empty.

Synchronize Shared Memory (cont.) SYNC

Restrictions:

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

Operation:

```
    SyncOperation(stype)
```

Exceptions:

None

System Call SYSCALL

31 26	25 6	5 0
SPECIAL 000000	code	SYSCALL 001100
6	20	6

Format: `SYSCALL` **MIPS32**

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 000000	rs	rt	code	TEQ 110100
6	5	5	10	6

Format: `TEQ rs, rt` **MIPS32**

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 000001	rs	TEQI 01100	immediate
6	5	5	16

Format: `TEQI rs, immediate` **MIPS32**

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 000000	rs	rt	code	TGE 110000
6	5	5	10	6

Format: `TGE rs, rt` **MIPS32**

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

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TGEI 01000	immediate
6	5	5	16

Format: `TGEI rs, immediate` **MIPS32**

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 000001	rs	TGEIU 01001	immediate
6	5	5	16

Format: `TGEIU rs, immediate` **MIPS32**

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 000000	rs	rt	code	TGEU 110001
6	5	5	10	6

Format: `TGEU rs, rt` **MIPS32**

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 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBP 001000
6	1	19	6

Format: `TLBP` **MIPS32**

Purpose:

To 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:

None

Operation:

```
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]$_R$ = EntryHi$_R$) and
      ((TLB[i]$_G$ = 1) or (TLB[i]$_{ASID}$ = EntryHi$_{ASID}$)) then
      Index ← i
   endif
endfor
```

Exceptions:

Coprocessor Unusable

Read Indexed TLB Entry TLBR

31 26	25	24 6	5 0
COP0	CO	0	TLBR
010000	1	000 0000 0000 0000 0000	000001
6	1	19	6

Format: `TLBR` **MIPS32**

Purpose:

To 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 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.

Restrictions:

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.

Read Indexed TLB Entry TLBR

Operation:

```
i ← Index
if i > (TLBEntries - 1) then
   UNDEFINED
endif
$PageMask_{Mask} \leftarrow TLB[i]_{Mask}$
$EntryHi \leftarrow TLB[i]_{R} \,||\, 0^{Fill} \,||$
         $TLB[i]_{VPN2} \,||$
         $0^{5} \,||\, TLB[i]_{ASID}$
$EntryLo1 \leftarrow 0^{Fill} \,||$
         $TLB[i]_{PFN1} \,||$
         $TLB[i]_{C1} \,||\, TLB[i]_{D1} \,||\, TLB[i]_{V1} \,||\, TLB[i]_{G}$
$EntryLo0 \leftarrow 0^{Fill} \,||$
         $TLB[i]_{PFN0} \,||$
         $TLB[i]_{C0} \,||\, TLB[i]_{D0} \,||\, TLB[i]_{V0} \,||\, TLB[i]_{G}$
```

Exceptions:

Coprocessor Unusable

Write Indexed TLB Entry TLBWI

31 26	25	24 6	5 0
COP0	CO	0	TLBWI
010000	1	000 0000 0000 0000 0000	000010
6	1	19	6

Format: `TLBWI` **MIPS32**

Purpose:

To 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. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in 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:

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.

Write Indexed TLB Entry TLBWI

Operation:

```
i ← Index
$TLB[i]_{Mask}$ ← $PageMask_{Mask}$
$TLB[i]_{R}$ ← $EntryHi_{R}$
$TLB[i]_{VPN2}$ ← $EntryHi_{VPN2}$
$TLB[i]_{ASID}$ ← $EntryHi_{ASID}$
$TLB[i]_{G}$ ← $EntryLo1_{G}$ and $EntryLo0_{G}$
$TLB[i]_{PFN1}$ ← $EntryLo1_{PFN}$
$TLB[i]_{C1}$ ← $EntryLo1_{C}$
$TLB[i]_{D1}$ ← $EntryLo1_{D}$
$TLB[i]_{V1}$ ← $EntryLo1_{V}$
$TLB[i]_{PFN0}$ ← $EntryLo0_{PFN}$
$TLB[i]_{C0}$ ← $EntryLo0_{C}$
$TLB[i]_{D0}$ ← $EntryLo0_{D}$
$TLB[i]_{V0}$ ← $EntryLo0_{V}$
```

Exceptions:

Coprocessor Unusable

Write Random TLB Entry TLBWR

31 26	25	24 6	5 0
COP0	CO	0	TLBWR
010000	1	000 0000 0000 0000 0000	000110
6	1	19	6

Format: `TLBWR` **MIPS32**

Purpose:

To 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. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- 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.

Restrictions:

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.

Write Random TLB Entry TLBWR

Operation:

```
$i \leftarrow Random$
$TLB[i]_{Mask} \leftarrow PageMask_{Mask}$
$TLB[i]_{R} \leftarrow EntryHi_{R}$
$TLB[i]_{VPN2} \leftarrow EntryHi_{VPN2}$
$TLB[i]_{ASID} \leftarrow EntryHi_{ASID}$
$TLB[i]_{G} \leftarrow EntryLo1_{G}$ and $EntryLo0_{G}$
$TLB[i]_{PFN1} \leftarrow EntryLo1_{PFN}$
$TLB[i]_{C1} \leftarrow EntryLo1_{C}$
$TLB[i]_{D1} \leftarrow EntryLo1_{D}$
$TLB[i]_{V1} \leftarrow EntryLo1_{V}$
$TLB[i]_{PFN0} \leftarrow EntryLo0_{PFN}$
$TLB[i]_{C0} \leftarrow EntryLo0_{C}$
$TLB[i]_{D0} \leftarrow EntryLo0_{D}$
$TLB[i]_{V0} \leftarrow EntryLo0_{V}$
```

Exceptions:

Coprocessor Unusable

Trap if Less Than TLT

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TLT 110010
6	5	5	10	6

Format: `TLT rs, rt` **MIPS32**

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 000001	rs	TLTI 01010	immediate
6	5	5	16

Format: `TLTI rs, immediate` **MIPS32**

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 000001	rs	TLTIU 01011	immediate
6	5	5	16

Format: `TLTIU rs, immediate` **MIPS32**

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` **MIPS32**

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 000000	rs	rt	code	TNE 110110
6	5	5	10	6

Format: `TNE rs, rt` **MIPS32**

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 000001	rs	TNEI 01110	immediate
6	5	5	16

Format: `TNEI rs, immediate` **MIPS32**

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

Floating Point Truncate to Long Fixed Point TRUNC.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	TRUNC.L 001001
6	5	5	5	5	6

Format: `TRUNC.L.S fd, fs` **MIPS64**
`TRUNC.L.D fd, fs` **MIPS64**

Purpose:

To convert an FP value to 64-bit fixed point, rounding toward zero

Description: `fd ← convert_and_round(fs)`

The value in FPR *fs,* in format *fmt,* is converted to a value in 64-bit long fixed point format and rounded toward zero (rounding mode 1). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}-1$, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode.

Operation:

```
    StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Floating Point Truncate to Long Fixed Point (cont.) TRUNC.L.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Overflow, Inexact

Floating Point Truncate to Word Fixed Point — TRUNC.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	TRUNC.W 001101
6	5	5	5	5	6

Format: `TRUNC.W.S fd, fs` **MIPS32**
`TRUNC.W.D fd, fs` **MIPS32**

Purpose:

To convert an FP value to 32-bit fixed point, rounding toward zero

Description: `fd ← convert_and_round(fs)`

The value in FPR *fs*, in format *fmt*, is converted to a value in 32-bit word fixed point format using rounding toward zero (rounding mode 1). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the *FCSR*. If the Invalid Operation *Enable* bit is set in the *FCSR*, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}–1$, is written to *fd*.

Restrictions:

The fields *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed point; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format *fmt*; if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

Operation:

```
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Floating Point Truncate to Word Fixed Point (cont.) TRUNC.W.fmt

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Invalid Operation, Overflow, Unimplemented Operation

Enter Standby Mode **WAIT**

31 26	25	24 6	5 0
COP0	CO	Implementation-Dependent Code	WAIT
010000	1		100000
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 5K 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.

In order to ensure that the clocks are stopped no coprocessor instructions must be placed within four instructions after the WAIT instruction.

Enter Standby Mode (cont.) WAIT

Operation:

```
Enter  lower power mode
```

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` **MIPS32**

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 001110	rs	rt	immediate
6	5	5	16

Format: `XORI rt, rs, immediate` **MIPS32**

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

Appendix A

Revision History

Revision	Date	Description
02.04	January 15, 2001	Major update & release.
02.05	March 30, 2001	Changes for preliminar FPU instruction and register description
02.06	June 28, 2001	Update with EJTAG Fastdata feature. Added information about Floating-Point Unit featues in 5Kf core. Minor updates.
02.07	August 31, 2001	Update with Simple BE feature. Minor updates.
02.08	May 28, 2002	Fixed typos in section 2.4 Limited Dual Issue. Repeat Rates updated and updates in general to chapter 3, Floating-Point Unit.